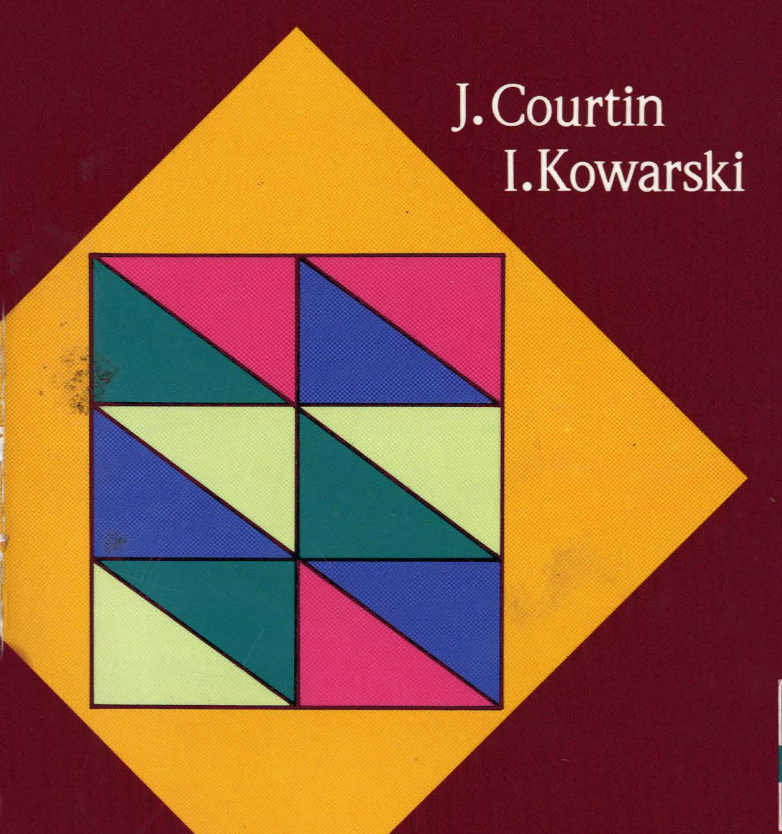
Initiation à l'algorithmique et aux structures de données

3. Problèmes, études de cas

J. Courtin
I. Kowarski



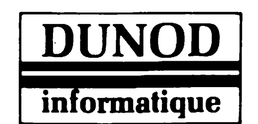
DUNOD
informatique

Initiation à l'algorithmique et aux structures de données

3. Problèmes, études de cas

JACQUES COURTIN
Professeur au département informatique de l'IUT 2 de Grenoble

IRÈNE KOWARSKI
Maître de conférences au département informatique de l'IUT 2 de Grenoble



ISBN : 2-04-019702-8

SOMMAIRE

AVANT - PROPOS

If you can solve it, it is an exercise ;
otherwise it's a research problem
R. Bellman, cité par D. Knuth
(Fundamental Algorithms, vol 1)

Ce troisième tome a pour objectif de mettre en œuvre les principes et les méthodes algorithmiques qui ont été développés dans les deux premiers tomes. Comme eux, il s'adresse aux étudiants de première année (DEUG, DEUST, IUT, BTS, MIAGE, MST, Licence, Écoles d'Ingénieurs...) et plus généralement à tous les lecteurs désireux de s'initier à la construction d'algorithmes corrects, qui sont la base de toute bonne programmation. Les algorithmes sont exprimés dans un langage proche de Pascal.

Comme le soulignait Jacques ARSAC dans la préface du premier tome : *Sidney Michaelson a comparé le rôle du professeur de programmation à celui du maître artisan qui formait ses compagnons. Pas de cours théorique. Le maître travaillait devant les élèves, analysant soigneusement ce qu'il faisait, expliquant chaque étape, justifiant chacune de ses décisions, soulignant les pièges à éviter... Peu à peu, l'élève s'appropriait les méthodes du maître artisan, devenant capable de création.* Les étudiants nous ont souvent demandé des exercices avec des propositions de solutions afin qu'ils puissent comme ils disent : "s'entraîner". Ce troisième tome devrait répondre à leur attente et leur permettre, petit à petit mais régulièrement, comme un athlète se prépare à une compétition sportive, de maîtriser les acquis fondamentaux de l'algorithmique indispensable à toute activité de programmation quel que soit le langage utilisé. Comme le disait si bien Boileau dans son "Art Poétique" :

> *Hâtez-vous lentement, et sans perdre courage*
> *Vingt fois sur le métier remettez votre ouvrage.*
> *Polissez-le sans cesse et le repolissez.*

Dans ce troisième tome, il s'agit de problèmes non réels mais qui pourraient être la modélisation simplifiée de situations concrètes fréquemment rencontrées dans la réalité. Nous nous sommes attachés à résoudre ces

problèmes en utilisant les techniques algorithmiques définies dans les précédents tomes. Très souvent, il sera demandé au lecteur de transformer ou d'adapter un algorithme connu utilisant des variables simples (voir tomes 1 et 2) à des variables structurées.

L'adaptation de nos solutions à des énoncés traitant de gros volumes d'informations nécessiterait parfois l'adoption d'autres algorithmes afin de mieux tenir compte des performances que nous avons sciemment partiellement négligées afin de toujours proposer des solutions simples.

Étant donné que nous étions limités à environ 200 pages et que nous souhaitions donner le maximum d'exercices avec leurs solutions, nous n'avons pas systématiquement remis les raisonnements par récurrence. Néanmoins, nous invitons fortement le lecteur à les effectuer et à se reporter, si nécessaire, à ceux que nous avons donnés dans les tomes précédents.

Dans chaque problème, on devra utiliser les algorithmes spécifiés dans les questions précédentes, même si ceux-ci n'ont pas encore été entièrement réalisés.

Les algorithmes de dichotomie sont volontairement proposés plusieurs fois en raison des difficultés rencontrées par les étudiants notamment en ce qui concerne la formulation précise de la postcondition, d'où découlent la mise en place correcte de l'invariant et le raisonnement par récurrence.

Les exercices proposés sont de niveaux très divers afin que ce document soit utile au plus grand nombre. Tous les énoncés ont été testés sur les étudiants soit en travaux dirigés soit à l'occasion de contrôles de connaissances.

Nous tenons ici à remercier très chaleureusement tous les collègues qui nous ont encouragés et aidés dans la réalisation de ce troisième tome, en particulier, Marcel BOUHIER et Damien GENTHIAL qui nous ont proposé quelques exercices et fait des remarques toujours constructives.

1

LES COMPOSANTS ÉLÉMENTAIRES DES ALGORITHMES

1.1. Prédicats

Énoncés

1.
Une boulangerie est ouverte de 7 heures à 13 heures et de 16 heures à 19 heures, sauf le dimanche après-midi et le lundi toute la journée. Écrire le corps de la fonction :
fonction boulouverte (d heure : entier ; d jour : chaîne8) : booléen ;
spécification {0 ≤ **heure** ≤ 24} => {**boulouverte** prend la valeur **vrai** si et seulement si la boulangerie est ouverte au **jour** et à l'**heure** indiqués}

2.
Une année est bissextile si son millésime est multiple de 4, sauf les années de début de siècle qui ne sont bissextiles que si leur millésime est divisible par 400. Écrire le corps de la fonction :
fonction bissextile (d année : entier) : booléen ;
spécification {0 ≤ **année** ≤ 10000} => {**bissextile** prend la valeur **vrai** si et seulement si l'**année** est bissextile}

3.
Soit le type structuré **date**, formé de trois nombres entiers qui indiquent respectivement le jour, le mois et l'année :
type date = structure
jour, mois, année : entier
fin;
Écrire le corps de la fonction :
fonction avant (d d1, d2 : date) : booléen ;
spécification { } => {**avant** prend la valeur **vrai** si et seulement si la date **d1** est strictement antérieure à la date **d2**}

Solutions proposées

1.

fonction boulouverte (d heure : entier ; d jour : chaîne8) : booléen ;
spécification {0 ≤ **heure** ≤ 24} => {**boulouverte** prend la valeur **vrai** si et seulement si la boulangerie est ouverte au **jour** et à l'**heure** indiqués}

Cette fonction booléenne peut s'exprimer de très nombreuses façons. Ainsi, la combinaison de plusieurs tests peut s'obtenir soit à l'aide de "si-alors-sinon-si ..." soit à l'aide de "et" et "ou", solution qui nous paraît bien préférable. De plus, on peut soit effectuer des tests (si - sinon) et selon leurs résultats affecter **vrai** ou **faux** au booléen, soit affecter directement le résultat des tests au booléen. C'est cette dernière manière d'opérer qui donne les versions les plus concises.
Nous donnons trois versions possibles, de la plus compliquée à la plus simple (on pourrait en imaginer encore bien d'autres !).

Première version :

```
début
    si jour = 'lundi' alors boulouverte := faux
    sinon
        si jour = 'dimanche' alors
        début
            si (heure <7) ou (heure>13) alors boulouverte := faux
            sinon  boulouverte := vrai
        fin
        sinon
            si (heure<7) ou ((heure>13) et (heure<16))
                                                ou (heure>19) alors
                boulouverte := faux
            sinon  boulouverte := vrai
fin;
```

On notera que ce type d'écriture nécessite une grande attention au niveau des si-sinon, avec inclusion de début-fin pour éviter que le sinon ne se rapporte automatiquement au si immédiatement au-dessus.

Deuxième version :

```
début
    boulouverte := faux;
    si (jour ≠ 'lundi') et (heure≥7) et (heure≤13) alors
        boulouverte := vrai
    sinon
        si (jour ≠'dimanche') et (jour≠'lundi') et (heure≥16)
                                                    et (heure ≤19) alors
            boulouverte := vrai
fin;
```

Dans cette version, nous avons initialisé le résultat **boulouverte** avec une valeur "par défaut" égale à **faux** ; il suffit alors de modifier le résultat dans les seuls cas où il est égal à **vrai.**

Troisième version :

```
début
    boulouverte := ((jour ≠ 'lundi') et (heure≥7) et (heure≤13))
                   ou
                   ((jour ≠'dimanche') et (jour≠'lundi')
                              et (heure≥16) et (heure ≤19))
fin;
```

2.

fonction bissextile (d année : entier) : booléen ;
spécification {0 ≤ **année** ≤ 10000} => {**bissextile** prend la valeur **vrai** si et seulement si l'**année** est bissextile}

Selon les idées de la question ci-dessus, nous proposons à nouveau trois versions.

Première version :

```
début
    si (année mod 4) ≠ 0 alors bissextile := faux
    sinon {(année mod 4) =0}
        si ((année mod 100) = 0) et ((année mod 400) ≠ 0)
            alors bissextile = faux
        sinon {((année mod 4) =0) et
            (((année mod 100) ≠0) ou ((année mod 400) = 0))}
            bissextile := vrai
fin;
```

Deuxième version :

```
début
    bissextile := faux;
    si (année mod 4) = 0 alors
        si ((année mod 100) ≠ 0) ou ((année mod 400) = 0) alors
            bissextile := vrai
fin;
```

Troisième version :

```
début
    bissextile := ((année mod 4) = 0) et
                  (((année mod 100) ≠ 0) ou ((année mod 400) = 0))
fin;
```

3.

Dans cette question, il est nécessaire d'accorder la priorité à la comparaison des années, puis à celle des mois, et enfin à celle des jours.

fonction avant (d d1, d2 : date) : booléen ;
spécification { } => {**avant** prend la valeur **vrai** si et seulement si la date **d1** est strictement antérieure à la date **d2**}

Première version (avec si-sinon en cascade, et initialisation du résultat) :

```
début
    avant := faux;
    si date1.année < date2.année alors avant := vrai
    sinon
        si date1.année = date2.année alors
            si date1.mois < date2.mois alors avant := vrai
            sinon
                si date1.mois = date2.mois alors
                    si date1.jour < date2.jour alors avant := vrai
fin;
```

Deuxième version (affectation d'une expression booléenne composée) :

```
début
    avant := (date1.année < date2.année)
             ou
             (( date1.année = date2.année) et (date1.mois < date2.mois))
             ou
             (( date1.année = date2.année) et (date1.mois = date2.mois)
                                            et (date1.jour < date2.jour))
fin;
```

1.2. Algorithmes numériques

Énoncés

1.
On souhaite calculer le montant des impôts dûs par un contribuable en fonction de son revenu imposable **rimp** et de son nombre de "parts fiscales" **nbparts**.
Les "règles du jeu" sont les suivantes :
- l'impôt total est égal à **nbparts** fois l'impôt par part ;
- le revenu par part **rpart** est égal au quotient de **rimp** par **nbparts** ;
- l'impôt par part est calculé selon le barême :
    - 0 si **rpart** est inférieur à 25 000 F
    - 10% sur la tranche de **rpart** comprise entre 25 000 et 50 000 F
    - 25% sur la tranche de **rpart** comprise entre 50 000 et 100 000 F
    - 50% sur le **rpart** qui dépasse 100 000 F

On notera que tous les nombres utilisés devront être de type "réel" : les revenus et impôts parce qu'ils peuvent dépasser la capacité d'un nombre entier (le plus souvent, les entiers doivent être inférieurs à 32 768), le nombre de parts parce qu'il peut être fractionnaire (demi-parts pour enfants).

Écrire le corps de la fonction :
fonction impôt (d rimp , nbparts : réel) : réel;

2.
On considère la procédure BOF ci-dessous :

```
procédure bof (d i : entier; r val : entier) ;
spécification { } => { ??? }
var trouvé : booléen;
début
    trouvé := faux;
    tantque (i ≤25) et non trouvé faire
        si i = 25 alors trouvé := vrai
        sinon i := i+1;
    val := i
fin;
```

2.1. Donner la condition de sortie de l'itération, ainsi que le tableau de sortie .
2.2. En déduire la postcondition de la procédure **bof**.
2.3. Remplacer cette procédure par une autre plus simple, qui donne la même postcondition.

3. Calcul d'intérêts composés :
Un capital C placé à un taux annuel de i devient, après un délai de un an, le capital C(1 + i).
Exemple :

année 0 on place 1000 F à un taux de 10% (soit i = 0.1 , donc 1+i = 1.1)
année 1 ce capital est devenu 1000 *1.1 = 1100 F
année 2 ce capital est devenu 1100 *1.1 = 1210 F
année 3 ce capital est devenu 1210 *1.1 = 1331 F

Écrire l'algorithme suivant :
fonction capital (d n : entier; d cap, taux : réel) : réel;
spécification {**n** ≥0, **cap** ≥ 0, **taux** ≥ 0} => {**capital** = valeur du capital initial **cap** placé à **taux** , pendant **n** années}

4. Calcul d'une valeur approchée de la fonction exponentielle e^x.

On admettra que $$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!} + \dots$$

ou encore $$e^x = \sum_{i=0}^{\infty} a_i \qquad \text{avec} \qquad a_i = \frac{x^i}{i!}$$

On arrêtera le calcul dès que la valeur de a_i devient inférieure à une valeur donnée epsilon. Écrire la fonction d'en-tête :
fonction expo (d x , epsilon : réel) : réel;
spécification {**x** ≥ 0, 0< **epsilon** <1} => {**expo** = e^x, avec une précision de l'ordre de **epsilon**}

Solutions proposées

1.

```
fonction impôt (d rimp , nbparts : réel) : réel;
var rpart, impart : réel;
début
    {calcul du revenu par part}
    rpart := rimp / nbparts ;
    {calcul de l'impôt par part}
    si rpart ≤ 25 000 alors impart := 0
    sinon {rpart > 25 000}
        si rpart ≤ 50 000 alors impart := (rpart - 25 000) * 0.1
        sinon {rpart > 50 000}
            si rpart ≤ 100 000 alors
                impart := 2 500 + (rpart - 50 000) * 0.25
            sinon {rpart > 100 000}
                impart := 15 000 + (rpart - 100 000) * 0.5 ;
        impôt := impart * nbparts {calcul de l'impôt total}
fin;
```

2. Etude de la procédure BOF :

2.1. La condition de sortie de l'itération est égale à la négation de la condition sur laquelle porte l'itération :

non ((i≤25) et non trouvé) = non (i≤25)) ou (non non trouvé)
= (i > 25) ou trouvé

Le tableau de sortie se présente ainsi :

i > 25	trouvé	commentaires
vrai	vrai	impossible
vrai	faux	on n'est pas entré dans l'itération
faux	vrai	i = 25
faux	faux	impossible à la sortie de l'itération

Ligne 1 : cas impossible, en effet **trouvé** ne peut devenir **vrai** que si i = 25
Ligne 2 : la valeur initiale de i était > 25, on n'est donc pas entré dans l'itération et **trouvé** est resté **faux**
Ligne 3 :i = 25
Ligne 4 :la condition de sortie de l'itération n'est pas vérifiée

2.2. Les deux cas possibles à la sortie de l'itération sont ceux de :
- la ligne 2 : la valeur initiale de i était > 25, on ne la modifie pas
- la ligne 3 : la valeur initiale de i était ≤ 25, sa valeur finale est de 25

La variable "résultat" **val** prend la valeur finale de la "donnée" **i**, modifiée ou non par l'itération.

La postcondition peut donc s'écrire :

{(i> 25 , val = i) ou (i ≤ 25, val = 25)}

ou encore

{val = max (i, 25)}

2.3.

```
procédure BOF (d i : entier; r val : entier) ;
spécification { } => {val = max (i, 25)}
début
    si i >25 alors   val := i
    sinon   val := 25
fin;
```

3. Raisonnement par récurrence (très proche du raisonnement présenté Tome 1, p. 36, seules la précondition et donc les initialisations diffèrent) :
Supposons le problème partiellement résolu, avec

$$\{\mathbf{total = cap * (1 + taux)^k \quad , \quad k \le n}\}$$

Il y a deux possibilités :

. k = n

Alors total = cap * $(1 + \text{taux})^n$, le calcul est terminé

. k < n

Alors en exécutant les actions
total := total * (1 + taux) ; k := k + 1 ,
nous obtenons de nouveau $\{\mathbf{total = cap * (1 + taux)^k \quad , \quad k \le n}\}$

L'itération s'écrit donc :

```
tantque k < n faire
début
      total := total * (1 + taux) ;
      k := k + 1
fin;
```

et l'assertion $\{\mathbf{total = cap * (1 + taux)^k \quad , \quad k \le n}\}$ est un invariant à condition d'être vraie à l'entrée dans l'itération.

Initialisation : Il suffit d'écrire **total := cap; k := 0;** pour vérifier l'assertion ci-dessus quelle que soit la valeur de **n** $\ge$ **0.**

```
fonction capital (d n : entier; d cap, taux : réel) : réel;
spécification {n ≥0, cap ≥ 0, taux ≥ 0} => {capital = valeur du capital
                                  initial cap placé à taux , pendant n années}
var total : réel;
    k : entier;
début
      total := cap; k := 0;
      {total = cap * (1 + taux)^k  ,   k ≤ n}
      tantque k < n faire
      début
            total := total * (1 + taux) ;
            {total = cap * (1 + taux)^(k+1)  ,   k < n}
            k := k + 1
            {total = cap * (1 + taux)^k  ,   k ≤ n}
      fin;
      { k ≥ n, total = cap * (1 + taux)^k  ,   k ≤ n
                       ==> total = cap * (1 + taux)^n}
      capital := total
fin;
```

4. La relation de récurrence qui donne a_n est la suivante :

$a_0 := 1;$
$a_n := a_{n-1} * x / n;$

On peut en déduire un raisonnement par récurrence qui conduit à l'algorithme suivant :

fonction expo (d x , epsilon : réel) : réel;
spécification {x ≥ 0, 0< **epsilon** <1} => {**expo** = e^x,
avec une précision de l'ordre de **epsilon**}
var terme, val : réel;
n : entier ;
début
val := 1;
terme:=1;
n := 0;

$$\{val = \sum_{i=0}^{n} \frac{x^i}{i!} \;,\; \frac{x^n}{n!} \geq epsilon\}$$

tantque terme ≥ epsilon faire
début

$$\{val = \sum_{i=0}^{n} \frac{x^i}{i!} \;,\; \frac{x^n}{n!} \geq epsilon\}$$

n := n + 1;
terme := terme * x / n;
val := val + terme

$$\{val = \sum_{i=0}^{n} \frac{x^i}{i!}\}$$

fin;

$$\{val = \sum_{i=0}^{n} \frac{x^i}{i!} \;,\; \frac{x^n}{n!} < epsilon\}$$

expo := val
fin;

2

LES FICHIERS SÉQUENTIELS

2.1. Annuaire du personnel

Énoncé

On dispose d'un fichier qui contient certaines informations sur le personnel d'une entreprise : nom et prénom (15 caractères au plus), fonction dans l'entreprise (également 15 caractères au plus), ancienneté dans l'entreprise (nombre entier d'années), numéro de poste téléphonique (trois chiffres), et présence effective (un booléen sera mis à **vrai** les jours où la personne est présente, et à **faux** lorsque la personne est en congé). Le fichier sera trié dans l'ordre alphabétique sur les identités des personnes, supposées toutes différentes.

Exemple d'une partie du fichier :

	Dupont Marie	**Dupont Michel**	**Durand Jules**	
	Secrétaire	**Directeur**	**Analyste**	
.....	**3**	**20**	**10**	.....
	234	**007**	**543**	
	faux	**faux**	**vrai**	

On disposera des types suivants :

```
type
    personne =    structure
                      identité, fonction : chaîne15;
                      ancienneté : entier;
                      tel : chaîne3;
                      présent : booléen
                  fin;
    fperso =  fichier de personne;
```

Écrire les algorithmes suivants :

1.

fonction nbabsents (dr fp : fperso ; d fonct : chaîne15) : entier;
spécification { } => {**nbabsents** = nombre de personnes qui occupent la fonction **fonct** et qui sont actuellement en congé}

2.

procédure listeposte (dr fp : fperso; d num : chaîne3) ;
spécification { } => {affichage de l'identité de toutes les personnes qui disposent du même poste de téléphone, de numéro **num**}

3.

procédure retraite (dr fp : fperso; r ancmax , nbancmax : entier) ;
spécification { } => {**ancmax** = ancienneté des personnes qui sont dans l'entreprise depuis le plus grand nombre d'années, **nbancmax** = nombre de personnes qui ont cette ancienneté}

4.

fonction persfonct (dr fp : fperso; d fonct : chaîne15 ; r existe : booléen) : chaîne15;
spécification { } => {**existe** = il existe au moins une personne, présente dans l'entreprise, qui occupe la fonction **fonct** , **persfonct** = identité de la première personne qui répond aux conditions}

5.

fonction téléphone (dr fp : fperso; d ident: chaîne15 ; r existe : booléen) : chaîne3;
spécification {**fp** est trié sur les identités croissantes} => {**existe** = il existe une personne de nom **ident**, **téléphone** = numéro du poste de cette personne}

6.

procédure congé (dr fp : fperso; d ident : chaîne15 ; r fpbis : fperso ; r trouvé : booléen) ;
Cette procédure vérifie que la personne d'identité **ident** est bien dans le fichier **fp** et qu'elle est actuellement présente. Si ces conditions sont remplies, un nouveau fichier **fpbis** est créé où cette personne est notée comme "absente" et le booléen **trouvé** est mis à vrai. Sinon, **trouvé** est mis à faux et **fpbis** sera identique à **fp**.

7.
procédure partir (dr fp : fperso ; d ident : chaîne15 ; r fpbis : fperso ;
r trouvé : booléen) ;
Cette procédure vérifie si la personne d'identité **ident** est bien dans le fichier **fp**. Si oui, un nouveau fichier **fpbis** est créé où cette personne ne figure plus et le booléen **trouvé** est mis à vrai. Sinon, **trouvé** est mis à faux et **fpbis** sera identique à **fp**.

8.
procédure quiestlà (dr fp : fperso; r fptrav : fperso ; r fpvac : fident) ;
Cette procédure effectue l'éclatement du fichier initial **fp** en deux fichiers, **fptrav** pour les personnes qui ne sont pas en congé, et **fpvac** pour les autres. Ce dernier fichier ne contiendra que l'identité des personnes absentes :
type fident = fichier de chaîne15;

9.
procédure embauche (dr fp : fperso ; dr newfp : fperso2; r fpbis : fperso) ;
Cette procédure effectue l'interclassement de l'ancien fichier **fp** et du fichier **newfp** de personnes qui viennent d'entrer dans l'entreprise. Le fichier **newfp** contient seulement les renseignements sur l'identité, la fonction et le numéro de poste :
type fperso2 = fichier de structure
identité, fonction : chaîne15;
tel: chaîne3;
fin;
Il est trié, comme **fp**, sur les identités. Le résultat sera le fichier **fpbis,** du même type **fperso** que **fp,** également trié sur les identités ; les nouvelles personnes y seront toutes considérées comme présentes dans l'entreprise, et ayant l'ancienneté égale à zéro.

Solutions proposées

1. Il s'agit d'une application directe de l'algorithme qui compte le nombre d'occurrences d'une valeur donnée dans un fichier (Tome 1, exercice 6, p. 222). Au lieu d'une valeur d'élément, on s'intéresse ici à la valeur d'une partie de l'enregistrement.

fonction nbabsents (dr fp : fperso ; d fonct : chaîne15) : entier;
spécification { } => {**nbabsents** = nombre de personnes qui occupent la fonction **fonct** et qui sont actuellement en congé}
var i : entier;

```
début
    i :=0 ;
    relire (fp) ;
    tantque non fdf (fp) faire
    début
        si non fp ↑. présent alors i := i + 1;
        prendre (fp)
    fin;
    nbabsents := i
fin;
```

2. L'algorithme est très proche du précédent, en remplaçant le comptage d'éléments par l'affichage, et en modifiant le critère de sélection.

```
procédure listeposte (dr fp : fperso; d num : chaîne3) ;
spécification { } => {affichage de l'identité de toutes les personnes qui
                      disposent du même poste de téléphone, de numéro num}
début
    relire (fp) ;
    tantque non fdf (fp) faire
    début
        si fp ↑. numéro = num alors
            écrire (fp ↑. identité) ;
        prendre (fp)
    fin;
fin;
```

3. Cet algorithme est proche de la procédure qui calcule l'élément maximum d'un fichier (Tome 1, p. 64). Il y a deux différences principales :
- on peut initialiser le maximum à 0, qui est un minorant évident, ce qui permet d'éviter le traitement séparé du premier enregistrement du fichier
- on ajoute un comptage à ancienneté égale : le compteur nmax est incrémenté à chaque rencontre du maximum amax, et réinitialisé à 1 à chaque changement de valeur de amax.

```
procédure retraite (dr fp : fperso; r ancmax , nbancmax : entier) ;
spécification { } => {ancmax = ancienneté des personnes qui sont dans
                          l'entreprise depuis le plus grand nombre d'années ,
                 nbancmax = nombre de personnes qui ont cette ancienneté}
var amax , nmax : entier;
début
    relire (fp) ;
    amax := 0;
    nmax := 0;
    tantque non fdf (fp) faire
    début
        si fp ↑. ancienneté = amax alors nmax := nmax + 1
```

```
        sinon
            si fp ↑. ancienneté > amax  alors
            début
                amax := fp ↑. ancienneté ;
                nmax := 1
            fin;
        prendre (fp)
    fin;
    ancmax := amax ;
    nbancmax := nmax
fin;
```

4. Le critère de recherche est la fonction, alors que le fichier est trié sur les identités ; cet algorithme est donc une application directe de la recherche associative dans un fichier non trié (Tome 1, p. 74).

```
fonction persfonct (dr fp : fperso; d fonct : chaîne15 ; r existe : booléen)
                                                              : chaîne15;
spécification  { } => {existe  = il existe au moins une personne, présente
                         dans l'entreprise, qui occupe la fonction fonct  ,
   persfonct = identité de la première  personne qui répond aux conditions}
var trouvé: booléen;
début
    trouvé := faux ;
    relire (fp) ;
    tantque non fdf (fp)  et non trouvé faire
        si  (fp ↑. fonction = fonct)  et  fp ↑. présent  alors
        début
            persfonct := fp ↑. identité ;
            trouvé := vrai
        fin
        sinon
            prendre (fp);
    existe := trouvé
fin;
```

5. Cette fois le critère de recherche est l'identité, l'algorithme est donc une application directe de la recherche associative dans un fichier trié (Tome 1, p. 82).

```
fonction  téléphone (dr fp : fperso; d ident: chaîne15 ;
                                        r existe : booléen) : chaîne3;
spécification  {fp est trié sur les identités croissantes} => {existe  = il existe
 une personne de nom ident, téléphone = numéro du poste de cette personne}
var   infer: booléen;
```

```
début
    existe := faux ;
    infer:= vrai;
    relire (fp) ;
    tantque non fdf (fp)  et infer faire
        si  ident > fp ↑. identité alors
            prendre (fp)
        sinon
            infer := faux;
    {fdf(fp)  v  (ident ≤fp ↑. identité)}
    si non fdf (fp) alors
        si ident = fp ↑. identité  alors
        début
            téléphone := fp ↑. tel;
            existe := vrai
        fin
fin;
```

On peut aussi envisager de traiter séparément, dans l'itération, les trois cas possibles: élément inférieur, égal ou supérieur. On obtient alors un algorithme tel que :

```
fonction téléphone (dr fp : fperso; d ident: chaîne15 ;
                                        r existe : booléen) : chaîne3;
spécification  {fp est trié sur les identités croissantes} => {existe  = il existe
une personne de nom ident, téléphone = numéro du poste de cette personne}
début
    existe := faux ;
    infer:= vrai;
    relire (fp) ;
    tantque non fdf (fp)  et infer faire
        si  ident > fp ↑. identité alors
            prendre (fp)
        sinon
            si ident = fp ↑. identité  alors
            début
                téléphone := fp ↑. tel;
                infer := faux;
                existe := vrai
            fin
            sinon infer := faux;
fin;
```

6. Il s'agit ici de la recopie d'un fichier dans un autre (Tome 1, p. 86), avec mise à jour d'un élément.

```
procédure congé (dr fp : fperso; d ident : chaîne15 ; r fpbis : fperso ;
                                                    r trouvé : booléen) ;
```

```
var  infeg : booléen;
début
     trouvé := faux ;
     infeg := vrai;
     relire (fp) ;
     récrire (fpbis) ;
     tantque non fdf (fp)  et infeg faire
          si fp ↑. identité > ident alors
               infeg := faux
          sinon
          début
               fpbis↑:= fp ↑;
               si  fp ↑. identité = ident alors
               début
                    infeg := faux;
                    si fp ↑. présent  alors  {mise à jour d'un élément}
                    début
                         fpbis↑.présent := faux;
                         trouvé := vrai
                    fin;
               fin;
               mettre (fpbis);
               prendre (fp);
          fin;

     {recopie dans fpbis de tous les éléments  restants}
     tantque non fdf (fp)   faire
     début
          fpbis↑:= fp ↑; mettre (fpbis);
          prendre (fp);
     fin;
fin;
```

7. On retrouve l'algorithme de recopie donné ci-dessus, mais cette fois en "sautant" un élément.

```
procédure partir (dr fp : fperso ; d ident : chaîne15 ; r fpbis : fperso ;
                                                  r trouvé : booléen) ;
var  infeg : booléen;
début
     trouvé := faux ;
     infeg := vrai;
     relire (fp) ;
     récrire (fpbis) ;
     tantque non fdf (fp)  et infeg faire
          si fp ↑. identité > ident alors
               infeg := faux
```

```
        sinon
            si fp ↑. identité = ident alors
            début {élément "ident" trouvé, ne pas le recopier}
                trouvé := vrai;
                infeg := faux;
                prendre (fp)
            fin
            sinon
            début {élément à recopier dans fpbis}
                fpbis ↑:= fp ↑; mettre (fpbis) ;
                prendre (fp)
            fin ;

    {recopie dans fpbis de tous les éléments  restants}
    tantque non fdf (fp)   faire
    début
        fpbis↑:= fp ↑; mettre (fpbis);
        prendre (fp);
    fin;
fin;
```

8. C'est l'algorithme d'éclatement sur un critère booléen (Tome 1, p. 89). Les deux fichiers de sortie n'ont pas la même structure, il faudra donc des traitements différents.

```
procédure quiestlà (dr fp : fperso; r fptrav : fperso ;  r fpvac : fident) ;
début
    relire (fp);
    récrire (fptrav) ;  récrire (fpvac) ;
    tantque non fdf (fp) faire
    début
        si fp↑. présent  alors
        début {personne présente, recopie dans fptrav}
            fptrav ↑:=  fp↑;
            mettre (fptrav)
        fin
        sinon
        début {personne absente, recopie de son identité dans fpvac}
            fpvac := fp ↑. identité ;
            mettre (fpvac)
        fin;
        prendre (fp)
    fin;
fin;
```

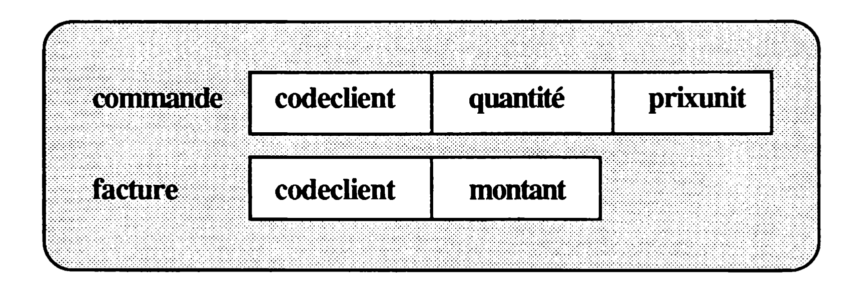
9. Nous retrouvons ici l'interclassement (Tome 1, p. 97), mais les deux fichiers à interclasser n'ont pas tout à fait la même structure. Il faudra donc traiter de manière différente les éléments correspondants. On supposera, pour simplifier, que l'on ne trouve pas dans fp et newfp de personnes ayant la même identité.

```
procédure embauche (dr fp : fperso ; dr newfp : fperso2; r fpbis : fperso) ;
début
    relire (fp); relire (newfp); récrire (fpbis);
    {parcours parallèle des deux fichiers, avec interclassement}
    tantque non fdf (fp) et non fdf (newfp) faire
        si fp↑. identité  <  newfp ↑. identité alors
        début {rangement dans fpbis de l'élément de fp}
            fpbis ↑:= fp ↑;
            mettre (fpbis) ; prendre (fp)
        fin
        sinon
        début {rangement dans fpbis de l'élément de  newfp}
            fpbis ↑. identité := newfp ↑. identité ;
            fpbis ↑. fonction := newfp ↑. fonction ;
            fpbis ↑. tel :=  newfp ↑. tel ;
            fpbis ↑. ancienneté := 0;
            fpbis ↑. présent := vrai;
            mettre (fpbis) ;
            prendre (newfp)
        fin;
    {recopie de la fin de fp}
    tantque non fdf (fp) faire
    début
        fpbis ↑:= fp ↑;
        mettre (fpbis) ;
        prendre (fp)
    fin;
    {recopie de la fin de  newfp}
    tantque non fdf (newfp) faire
    début
        fpbis ↑. identité := newfp ↑. identité ;
        fpbis ↑. fonction := newfp ↑. fonction ;
        fpbis ↑. tel :=  newfp ↑. tel ;
        fpbis ↑. ancienneté := 0;
        fpbis ↑. présent := vrai;
        mettre (fpbis) ;
        prendre (newfp)
    fin;
```

2.2. Facturation

Énoncé

Il s'agit ici de simuler une petite facturation très simplifiée d'une entreprise de vente par correspondance. Chaque client émet un ou plusieurs bons de commande. A partir de ces bons, on désire établir une ou plusieurs factures par client. Dans un premier temps, on dispose d'un fichier de type commandes contenant tous les bons de commande d'un ou de plusieurs clients. Une commande est composée d'un code client **codeclient**, de la quantité commandée **quantité** et du prix unitaire de l'article commandé **prixunit**. Une facture est composée du code client **codeclient** et du montant de la facture **montant**.



On utilisera les déclarations suivantes :

```
type
commande =  structure
                codeclient, quantité : entier;
                prixunit : réel
            fin;
facture  =  structure
                codeclient : entier;
                montant : réel
            fin;
commandes = fichier de commande;
factures  = fichier de facture;
```

1. PARCOURS DE FICHIERS

1.1. On souhaite vérifier que le fichier com est trié sur le code client. Écrire une

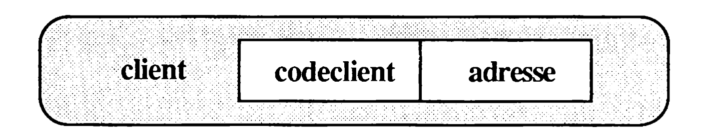
fonction trié (dr com: commandes) : booléen;
spécification { } => {**trié** = **com** est **trié** sur **com↑.codeclient**}

1.2. Si un client a commandé n articles, on dispose de n bons de commande pour ce client. On désire établir une facture pour chaque bon. Le fichier **com** n'est pas trié. Écrire une
procédure facturation1 (dr com : commandes ; r fact : factures);
spécification { } => {**fact** contient toutes les factures établies pour chaque bon de commande}

1.3. On suppose maintenant que le fichier de commandes **com** est trié par ordre croissant sur le codeclient et on souhaite maintenant connaître le nombre de clients différents présents dans **com**. Écrire une
fonction nbclient (dr com : commandes) : entier ;
spécification {**com trié** sur **com↑.codeclient**} =>
{**nbclient** = nombre de clients présents dans **com**}

1.4. Le fichier **com** est toujours trié, et on veut établir une seule facture pour tous les bons de commande d'un même client. Écrire une
procédure facturation2 (dr com : commandes ; dr fact : factures);
spécification {**com trié** sur **com↑.codeclient**} =>
{une seule facture par client a été établie}

1.5. On dispose maintenant d'un fichier **clients** indiquant pour chaque client son **adresse.** Ce fichier est trié sur le **codeclient.**

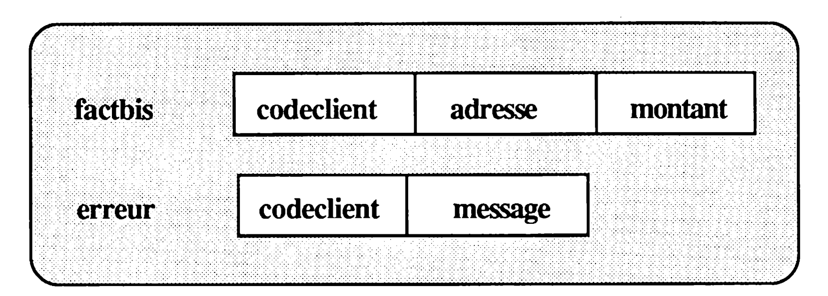


On utilisera les déclarations supplémentaires suivantes :
client = structure
codeclient : entier;
adresse : chaîne50
fin;
clients = fichier de client;

Écrire une
procédure adresse (dr cli : clients; codeclient : entier; r adr : chaîne; r trouvé : booléen);
spécification {**cli trié** sur **cli↑.codeclient**} => {**(fdf(cli)** v **(cli↑.codeclient ≠ codeclient) , non trouvé)**
v **(cli↑.codeclient = codeclient , adr = cli↑.adresse , trouvé)**}

N.B. Les premiers éléments de cli ont déjà été parcourus à l'appel de adresse; **cli** est donc déjà ouvert en lecture. Cette procédure recherche l'adresse du client possédant le code **codeclient**. L'algorithme doit tenir compte du fait que le fichier **cli** est trié sur le code client.

1.6. On dispose maintenant des structures suivantes :



correspondant aux déclarations suivantes :

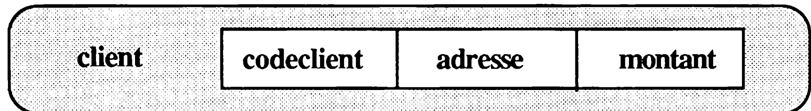
```
type
factbis   =   structure
                  codeclient : entier;
                  adresse : chaîne50;
                  montant : réel
              fin;
erreur    =   structure
                  codeclient : entier;
                  message : chaîne20
              fin ;

commandes =   fichier de commande;
factures  =   fichier de facture;
erreurs   =   fichier de erreur;
clients   =   fichier de client;
facturesbis = fichier de factbis;
```

Écrire une
procédure clientsuivant (dr com : commandes);
spécification {**com** trié sur **com↑.codeclient** , c = **com↑.codeclient**} =>
{(**fdf(com)** ∨(**com↑.codeclient** > c)}
qui recherche la première commande du prochain client sachant que **com** est déjà ouvert en lecture.

1.7. Écrire une
procédure facturation3 (dr com : commandes; dr cli : clients;
r fact : facturesbis; r err : erreurs);
spécification {**com trié** sur **com↑.codeclient** ,**cli trié** sur **cli↑.codeclient**} =>
{**fact** contient les factures établies pour chaque client, **err** contient les messages d'erreurs correspondant aux clients de **com** non présents dans **cli.**}
Cette procédure a pour objet d'établir une seule facture pour toutes les commandes d'un même client. L'algorithme doit tenir compte du fait que les fichiers **com** et **cli** sont triés. On utilisera les deux procédures précédentes.

1.8. On souhaite maintenant détecter les bons clients afin de leur faire un cadeau pour les fêtes de fin d'année. Pour cela on doit disposer du montant de toutes les commandes. On dispose de la structure suivante :



correspondant au nouveau type client défini comme suit :

```
client  =  structure
                codeclient : entier;
                adresse : chaîne50 ;
                montant : réel ;
           fin;
```

Écrire une
procédure facturation4 (dr com : commandes; dr cli : clients;
r ncli : clients; r fact : facturesbis; r err : erreurs);
spécification {**com trié** sur **com↑.codeclient,cli trié** sur **cli↑.codeclient**} =>
{**fact** contient les factures établies pour chaque client,
err contient les messages d'erreurs correspondant aux clients de **com** non présents dans **cli**,
ncli fichier mis à jour du fichier **cli**}

Solutions proposées

1.1. On utilise l'algorithme trié (Tome 1, p. 79) en l'adaptant aux variables structurées

```
fonction trié (dr com: commandes) : booléen;
spécification { } => {trié = com est trié sur com↑.codeclient}
var tri : booléen;  codeclientprécédent : entier;
début
    tri := vrai;  relire(com);
    si non fdf(com) alors
    début
        codeclientprécédent := com↑.codeclient;
        prendre (com);
        tantque  non fdf(com)  et tri faire
            si codeclientprécédent ≤ com↑.codeclient alors
            début
                codeclientprécédent := com↑.codeclient;
                prendre (com)  {lecture de l'élément courant}
            fin
            sinon  tri := faux
    fin;
    trié := tri
fin;
```

1.2. Il s'agit ici d'une simple création de fichiers à partir d'un autre. On utilise l'algorithme **copie** (Tome 1, p. 86) en l'adaptant au problème posé.

```
procédure facturation1 (dr com : commandes ; r fact : factures);
spécification { } => {fact contient toutes les factures établies pour chaque
                                                      bon de commande}
début
    relire (fcom);
    récrire (fact);
    tantque non fdf (fcom) faire
    début
        fact↑.codeclient := fcom↑.codeclient;
        fact↑.montant := fcom↑.quantité * fcom↑.prixunit;
        mettre (fact);
        prendre (fcom)
    fin
fin;
```

1.3. C'est un parcours classique d'un fichier trié avec comptage d'éléments dès que deux éléments consécutifs sont différents.

```
fonction nbclient (dr com : commandes) : entier ;
spécification {com trié sur com↑.codeclient} =>
                              {nbclient = nombre de clients présents dans com}
var codeclientprécédent, nb : entier;
début
    nb := 0;
    relire (com);
    si non fdf (com) alors
    début {fichier non vide}
        codeclientprécédent := com↑.codeclient;
        nb := 1;
        prendre (com);
        tantque non fdf (com) faire
        début
            si codeclientprécédent ≠ com↑.codeclient alors
            début {nouveau code client}
                nb := nb + 1;
                codeclientprécédent := com↑.codeclient;
            fin;
            prendre (com)
        fin;
    fin;
    nbclient := nb;
fin;
```

1.4. Première solution : Le fichier com est le fichier pilote. Il n'y a qu'une seule itération.

```
procédure facturation2 (dr com : commandes ; dr fact : factures);
spécification {com trié sur com↑.codeclient} =>
                                {une seule facture par client a été établie}
var montantclient : réel;  codeclient : entier;
début
    relire (com);
    récrire (fact);
    si non fdf (com) alors
    début  {fichier non vide}
        {initialisation pour le premier client}
        montantclient := com↑.quantité * com↑.prixunit;
        codeclient := com↑.codeclient;
        prendre (com);
        tantque non fdf (com) faire
        début
            si codeclient ≠ com↑.codeclient alors
            début  {nouveau client}
                { calcul de la facture pour l'ancien client}
                fact↑.codeclient := codeclient;
                fact↑.montant := montantclient;
                mettre (fact);
                {initialisation pour le  nouveau client}
                montantclient := com↑.quantité * com↑.prixunit;
                codeclient := com↑.codeclient;
            fin
            sinon  {même client , donc cumul de facture}
                montantclient := montantclient +
                                         com↑.quantité * com↑.prixunit;
            prendre (com);
        fin;
    fin;
    {calcul de la facture pour le dernier client}
    fact↑.codeclient := codeclient;
    fact↑.montant := montantclient;
    mettre (fact);
fin;
```

Deuxième solution : A l'intérieur de l'itération sur le fichier com, il y a une deuxième itération pour parcourir toutes les commandes d'un même client.

```
procédure facturation2 (dr com : commandes ; dr fact : factures);
spécification {com trié sur com↑.codeclient} =>
                                {une seule facture par client a été établie}
var  montantclient : réel;  codeclient : entier;  mêmeclient : booléen;
```

```
début
      relire(com);  récrire(fact);
      tantque non fdf(com) faire
      début
            {initialisation pour la première commande du client}
            mêmeclient := vrai;  montantclient := 0.0;
            codeclient := com↑.codeclient;
            tantque mêmeclient  et  non fdf(com)  faire
            début {toujours le même client}
                  montantclient := montantclient +
                                          com↑.quantite * com↑.prixunit;
                  prendre(com); {on passe à la commande suivante}
                  si non fdf(com) alors
                        mêmeclient := codeclient = com↑.codeclient;
            fin;
            {mise en place de la facture pour toutes les commandes d'un client}
            fact↑.codeclient := codeclient;  fact↑.montant := montantclient;
            mettre(fact);
      fin;
fin;
```

1.5.

```
procédure adresse (dr cli : clients; codeclient : entier; r adr : chaîne;
                                                            r trouvé : booléen);
spécification  {cli trié sur cli↑.codeclient} => {(fdf(cli)
                              ∨(cli↑.codeclient ≠ codeclient) , non trouvé)
            ∨(cli↑.codeclient = codeclient , adr = cli↑.adresse , trouvé)}
{les premiers éléments de cli ont déjà été parcourus à l'appel de adresse}
var  infer : booléen;
début
      infer := vrai;
      trouvé := faux;
      tantque   non fdf(cli)  et  infer faire
            si cli↑.codeclient < codeclient alors
                  prendre(cli)
            sinon
            début
                  {cli↑.codeclient ≥ codeclient}
                  infer := faux;
                  si cli↑.codeclient = codeclient alors
                  début
                        adr := cli↑.adresse;
                        trouvé := vrai
                  fin;
            fin;
fin;
```

1.6.

```
procédure clientsuivant (dr com : commandes);
spécification  {com trié sur com↑.codeclient , c = com↑.codeclient} =>
                                        {(fdf(com) v(com↑.codeclient > c)}
var  égal : booléen;  codeclient : entier;
début
    égal := vrai;
    codeclient := com↑.codeclient;
    prendre(com);
    tantque  non fdf(com)  et  égal faire
        si com↑.codeclient = codeclient alors
            prendre(com)
        sinon
            égal := faux;
fin;
```

1.7. Première solution : On recherche d'abord le premier client. Ensuite on effectue une itération sur les deux fichiers **com** et **cli**.

```
procédure facturation3 (dr com : commandes;  dr cli : clients;
                                        r fact : facturesbis; r err : erreurs);
spécification  {com trié sur com↑.codeclient ,cli trié sur cli↑.codeclient} =>
                        {fact contient les factures établies pour chaque client,
                          err contient les messages d'erreurs correspondant
                                aux clients de com non présents dans cli.}
const  mess = 'client inconnu';
var  montantclient : réel;  codeclient : entier;
    prochainclient, premierclient : booléen ;  adr : chaîne50
début
    relire(com);  récrire(fact);
    récrire(err);  relire(cli);
    premierclient := faux;
    {recherche de l'adresse du premier client}
    tantque  non fdf(com)  et  non fdf(cli)  et  non premierclient faire
    début
        codeclient := com↑.codeclient;
        adresse (cli, codeclient , adr, premierclient);
        si non premierclient alors
        début
            {mise à jour du fichier err}
            err↑.codeclient := codeclient;
            err↑.message := mess;
            mettre(err);
            {l'adresse n'a pas été trouvée, on passe au client suivant}
            clientsuivant(com);
        fin;
    fin;
```

```
si premierclient alors
début
     {on a trouvé l'adresse du client}
     montantclient := com↑.quantité * com↑.prixunit;
     prendre (com);
     tantque non fdf(com) et non fdf(cli) faire
     début
          si codeclient = com↑.codeclient alors
          début
               {même client, on calcule le montant de sa facture
                              et on passe à la commande suivante}
               montantclient := montantclient +
                                   com↑.quantité * com↑.prixunit;
               prendre(com);
          fin
          sinon
          début
               {nouveau client, mise à jour de fact
                                   avec la facture du client précédent}
               fact↑.codeclient := codeclient;
               fact↑.montant := montantclient; fact↑.adresse := adr;
               mettre(fact); prochainclient := faux;
               tantque non fdf(com) et non fdf(cli)
                                        et non prochainclient faire
               début {recherche de l'adresse du client suivant}
                    codeclient := com↑.codeclient;
                    adresse(cli, codeclient, adr, prochainclient);
                    si non prochainclient alors
                    début {codeclient ∉ cli , mise à jour de err}
                         err↑.codeclient := codeclient;
                         err↑.message := mess; mettre(err);
                         clientsuivant(com);
                         {première commande du prochain client}
                    fin
                    sinon
                    début
                        {codeclient ∈ cli , on initialise le montant de la
                           facture et on passe à la commande suivante}
                         montantclient :=
                                   com↑.quantité * com↑.prixunit;
                         prendre(com);
                    fin;
               fin;
          fin
     fin;
fin;
```

```
    si prochainclient alors
    début {facture du dernier client}
        fact↑.codeclient := codeclient; fact↑.montant := montantclient;
        fact↑.adresse := adr; mettre(fact);
    fin;
    tantque non fdf(com) faire
    début {mise à jour de err avec les clients restant de com}
        err↑.codeclient := com↑.codeclient;
        err↑.message := mess; mettre(err);
        clientsuivant(com);
    fin;
fin;
```

Deuxième solution : Au départ, on confond le premier client et le client suivant, ce qui évite d'écrire des actions spécifiques pour rechercher le premier client. Ensuite, comme dans facturation2b, on itère sur le même client à l'intérieur de l'itération portant sur le fichier com.

```
procédure facturation3 (dr com : commandes; dr cli : clients;
                                    r fact : facturesbis; r err : erreurs);
spécification {com trié sur com↑.codeclient ,cli trié sur cli↑.codeclient} =>
                        {fact contient les factures établies pour chaque client,
                          err contient les messages d'erreurs correspondant
                                aux clients de com non présents dans cli.}
const mess = 'client inconnu';
var montantclient : réel; codeclient : entier;
    mêmeclient, prochainclient : booléen; adr : chaîne50;
début
    relire(com); récrire(fact); récrire(err); relire(cli);
    tantque non fdf(com) faire
    début {recherche client suivant}
        prochainclient := faux;
        tantque (non fdf(com)) et non prochainclient faire
        début
            {recherche de l'adresse du client suivant}
            codeclient := com↑.codeclient;
            adresse(cli, codeclient, adr, prochainclient);
            si non prochainclient alors
            début {codeclient n'est pas dans cli}
                {mise à jour de err}
                err↑.codeclient := codeclient;
                err↑.message := mess;
                mettre(err);
                {on cherche la première commande du prochain client}
                clientsuivant(com);
            fin
        fin;
```

```
        si prochainclient alors
        début
            mêmeclient := vrai;
            montantclient := 0.0;
            tantque mêmeclient et non fdf(com) faire
            début  {même client, on calcule le montant de sa facture
                                  et on passe à la commande suivante}
                montantclient := montantclient +
                                      com↑.quantite * com↑.prixunit;
                prendre(com);
                si non fdf(com) alors
                    mêmeclient := codeclient = com↑.codeclient
            fin;
            {mise à jour de fact avec la facture du client précédent}
            fact↑.codeclient := codeclient;
            fact↑.montant := montantclient;  fact↑.adresse := adr;
            mettre(fact);
        fin;
    fin
fin;
```

1.8.

```
procédure facturation4 (dr com : commandes;  dr cli : clients;
                        r ncli  : clients; r fact : facturesbis; r err : erreurs);
spécification  {com trié sur com↑.codeclient,cli trié sur cli↑.codeclient} =>
                       {fact contient les factures établies pour chaque client,
                        err contient les messages d'erreurs correspondant
                              aux clients de com non présents dans cli,
                                 ncli fichier mis à jour du fichier cli}
const  mess = 'client inconnu';
var   montantclient : réel;  codeclient : entier;
    infer, trouvé, mêmeclient : booléen;  adr : chaîne50;
début
    relire(com);  récrire(fact);  récrire(err);  relire(cli);  récrire(ncli);
    tantque non fdf(com) faire
    début
        codeclient := com↑.codeclient;
        infer := vrai;
        tantque  non fdf(cli)  et  infer faire
            si cli↑.codeclient < codeclient alors
            début
                ncli↑ := cli↑;  mettre(ncli);
                prendre(cli)
            fin
            sinon
                infer := faux;
```

```
        si fdf(cli) alors
            trouvé := faux
        sinon
            trouvé := cli↑.codeclient = codeclient;
        si non trouvé alors
        début
            {mise à jour de err}
            err↑.codeclient := codeclient;
            err↑.message := mess;
            mettre (err);
            {on cherche la première commande du prochain client}
            clientsuivant (com);
        fin
        sinon
        début
            mêmeclient := vrai;
            montantclient := 0.0;
            tantque mêmeclient et non fdf(com) faire
            début
                {même client, on calcule le montant de sa facture
                                et on passe à la commande suivante}
                montantclient := montantclient
                                + com↑.quantite * com↑.prixunit;
                prendre(com);
                si non fdf(com) alors
                    mêmeclient := codeclient = com↑.codeclient
            fin;
            {mise à jour de fact avec la facture du client précédent}
            fact↑.codeclient := codeclient;
            fact↑.montant := montantclient;
            fact↑.adresse := cli↑.adresse;
            mettre(fact);
            cli↑.montant := cli↑.montant + montantclient
        fin;
    fin;
    tantque non fdf(cli) faire
    début
        ncli↑ := cli↑;
        mettre(ncli);
        prendre(cli)
    fin
fin;
```

3

LES VECTEURS

3.1. Stock de voitures

Énoncé

Un marchand d'automobiles d'occasion souhaite gérer son stock à l'aide de quelques procédures très simples : recherche d'une voiture déterminée ou d'un ensemble de voitures correspondant à un critère, ajout d'une nouvelle voiture, suppression d'une voiture vendue ou envoyée à la casse, tri du stock sur divers critères...

On se contentera, dans cet exercice, de l'écriture de certaines de ces procédures, variantes des algorithmes du cours sur les vecteurs. En effet, on supposera que le stock, de 100 voitures au plus, puisse être représenté à l'aide d'un tableau en mémoire ("vecteur d'enregistrements structurés"), et on ne se préoccupera pas ici de sa sauvegarde éventuelle dans un fichier disque. Les seules informations retenues pour chaque voiture seront son numéro et son année d'immatriculation, sa marque et son modèle, ainsi que son prix. Bien entendu, le numéro d'immatriculation sera différent pour chaque voiture.

Exemple d'une partie du vecteur (non trié) :

	1234XZ38	**3456WW69**	**9999AA38**	
	1990	**1987**	**1953**	
....	**Peugeot**	**Peugeot**	**Citroën**	.....
	205 GT	**604GLS**	**2CV**	
	60000	**100000**	**1000**	

Les déclarations de types utilisées seront les suivantes :

type voiture = structure
numéro: chaîne8;
année : entier ;
marque, modèle : chaîne10;
prix : réel
fin;
tabv = tableau [1..100] de voiture;

Écrire les algorithmes suivants :

1.

procédure choixmodèle (d V : tabv ; d n : entier;
d marque, modèle : chaîne10);
spécification {**n** ∈ [0..100] , **V** trié sur les **numéros** croissants}
=> {affichage de l'**année** et du **prix** de chacune des voitures
de la **marque** et du **modèle** demandés,
ou d'un message si ce modèle n'est pas disponible}

2.

procédure choixprix (d V : tabv ; d n : entier ; d prixinf, prixsup : réel);
spécification {**n** ∈ [0..100] , **V** trié sur les **prix** croissants}
=> {affichage de l'année, de la marque et du modèle de chacune des
voitures dont le prix est compris entre **prixinf** et **prixsup**,
ou bien d'un message si aucune voiture n'est disponible}

3.

procédure recherche (d V : tabv ; d n : entier ; d num : chaîne8 ;
r existe: booléen; r posit:entier);
spécification {**n** ∈ [0..100] , **V** trié sur les **numéros** croissants}
=> {(**existe, V[posit].numéro = num**)
∨ (**¬existe, V[posit-1].numéro<num<V[posit].numéro**)}

3.1. Méthode séquentielle
a) Raisonnement par récurrence
b) Algorithme

3.2. Méthode dichotomique
a) Raisonnement par récurrence
b) Algorithme

4.

procédure newauto (dr V : tabv; dr n : entier; d auto : voiture;
r new: booléen);
spécification {**n** ∈ [0..100] , **V** trié sur les **numéros** croissants}
=> {(**auto.numéro** ∉ **V[1..n]** , **auto** a été inséré dans **V, new**)
∨ (**auto.numéro** ∈ **V[1..n]** ,¬ **new**)}

5.
procédure casse (dr V : tabv; dr n : entier; d num : chaîne8; r old : booléen);
spécification {**n** ∈ [0..100], **V** trié sur les **numéros** croissants}
=> {**num** ∈ **V[1..n]** , la voiture a été supprimée dans **V**, **old**)
∨ (**num** ∉ **V[1..n]** , ¬ **old**)}

6.
procédure retrier (dr V: tabv; d n : entier);
spécification {n ∈ [0..100] , V trié sur les **numéros** croissants}
=> {V est trié sur les **marques** dans l'ordre alphabétique,
et à marque égale, sur les **prix décroissants**}

On emploiera la méthode du "tri bulles optimisé"

Solutions proposées

1. Le vecteur n'est pas trié sur le critère de choix (marque et modèle), il faudra donc parcourir tout le vecteur en examinant chaque enregistrement pour voir s'il correspond au critère de choix. Un booléen initialisé à **faux** et positionné à **vrai** dès que la marque et le modèle recherchés ont été trouvés permet l'écriture éventuelle du message d'absence.

procédure choixmodèle (d V : tabv ; d n : entier;
d marque, modèle : chaîne10);
spécification {**n** ∈ [0..100] , **V** trié sur les **numéros** croissants}
=> {affichage de l'**année** et du **prix** de chacune des voitures
de la **marque** et du **modèle** demandés,
ou d'un message si ce modèle n'est pas disponible}

```
var  i : entier;
     trouvé : booléen;
début
     écrireln ('Voitures  ', marque, modèle, ' :');
     trouvé := faux;
     pour i := 1 haut n faire
          si (V[i] . marque = marque) et (V[i] . modèle = modèle) alors
          début
               écrireln(V[i] . année, V[i] . prix);
               trouvé := vrai
          fin ;
     {trouvé ne sert qu'à contrôler l'affichage du message d'erreur}
     si non trouvé alors
          écrireln ('aucune voiture disponible ')
fin;
```

2. Cette fois, le critère de choix (le prix) est un critère de tri du vecteur. Il faudra donc parcourir le vecteur jusqu'à la rencontre de la première voiture ayant un prix supérieur ou égal à prixinf, puis imprimer les caractéristiques des voitures rencontrées tant que le prix demeure inférieur ou égal à prixsup. Pour des raisons de cohérence (problème de l'indice qui pourrait dépasser 100) ainsi que de lisibilité, on pourra employer deux booléens, **inf** et **sup**, pour délimiter la zone utile du vecteur. Un troisième booléen, **trouvé**, permettra de gérer l'écriture du message d'erreur.

Première version :

```
procédure choixprix (d V : tabv ;  d n : entier ;  d prixinf, prixsup : réel);
spécification {n ∈ [0..100] , V trié sur les prix croissants}
        => {affichage de l'année, de la marque et du modèle  de chacune des
                    voitures dont le prix est compris entre prixinf et prixsup,
                     ou bien d'un message si aucune voiture n'est disponible}
var   i : entier;
      inf, sup , trouvé: booléen;
début
      écrireln ('Voitures  de prix compris entre ',prixinf,' et ',prixsup,' :');
      inf := vrai ;  i := 1;
      tantque (i≤n) et  inf faire
            si V[i].prix ≥ prixinf alors
                  inf := faux
            sinon
                  i :=  i+1;
      {(V[i].prix ≥prixinf) ou  (i>n)}
      sup := faux;
      trouvé := faux ;
      tantque (i≤n) et  non sup faire
            si V[i].prix ≤ prixsup  alors
            début  {(V[i].prix ≥prixinf)  et (V[i].prix ≤prixsup))
                  écrireln(V[i]  . année, V[i]  . marque, V[i]  . modèle) ;
                  trouvé := vrai;
                  i := i+1
            fin
            sinon  {V[i].prix > prixsup)
                  sup := vrai;
      si non trouvé alors écrireln ('aucune voiture disponible ')
fin;
```

Deuxième version :

On peut faire l'"économie" du booléen **inf** en écrivant une seule boucle de parcours ; le test de la première boucle est alors inclus dans la seconde. Il faut noter que cette version est en réalité plus lourde : le test de la borne inférieure est effectué pour tous les enregistrements de la zone utile du vecteur.

```
procédure choixprix (d V : tabv ; d n : entier ; d prixinf, prixsup : réel);
spécification {n ∈ [0..100] , V trié sur les prix croissants}
        => {affichage de l'année, de la marque et du modèle de chacune des
                voitures dont le prix est compris entre prixinf et prixsup,
                ou bien d'un message si aucune voiture n'est disponible}
var  i : entier;
     sup , trouvé: booléen;
début
     écrireln ('Voitures de prix compris entre ',prixinf,' et ',prixsup' :');
     i := 1;
     sup := faux; trouvé := faux ;
     tantque (i≤n) et non sup faire
          si (V[i].prix >prixsup) alors sup := vrai
          sinon
          début {V[i].prix ≤prixsup}
               si (V[i].prix ≥ prixinf) alors
               début {prixinf ≤ V[i].prix ≤prixsup}
                    écrireln(V[i] . année, V[i] . marque, V[i] . modèle) ;
                    trouvé := vrai;
               fin;
               i := i+1
          fin ;
     si non trouvé alors écrireln ('aucune voiture disponible ')
fin;
```

3.

```
procédure recherche (d V : tabv ; d n : entier ; d num : chaîne8 ;
                                        r existe: booléen; r posit:entier);
spécification {n ∈ [0..100] , V trié sur les numéros croissants}
                                   => {(existe, V[posit].numéro = num)
                 v (¬existe, V[posit-1].numéro<num<V[posit].numéro)}
```

3.1. Méthode séquentielle

a) Raisonnement par récurrence

Hypothèse: **V[1..i-1].numéro<num≤V[n].numéro** avec **i≤n** et **n>0.**

V[1..i-1]	i .. n-1	n
<num	??	≥num

.i=n alors V[1..n-1].numéro<num≤V[n].numéro

donc **existe** := num=V[n].numéro et **posit** := n soit **posit** := i

.i<n

.. V[i].numéro<num alors i := i+1 permet de retrouver l'hypothèse

.. V[i].numéro≥num alors V[1..i-1].numéro<num≤V[i].numéro

donc **existe** := num=V[i].numéro et **posit** := i

Initialisation:

- traiter les cas particuliers n=0 et num >V[n].numéro, afin de vérifier l'hypothèse de récurrence
- i := 1

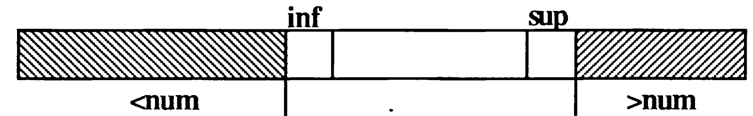
b) Algorithme

```
var i : entier;
début
    existe := faux;
    {initialisations}
    si n=0 alors
        posit := 1
    sinon
        si num > V[n].numéro alors
            posit := n+1
        sinon
        début {num ≤V[n].numéro}
            i := 1;
            tantque V[i].numéro<num faire
                i := i+1;
            {V[i].numéro ≥ num}
            si V[i].numéro=num alors
                existe := vrai;
            posit := i
        fin
fin;
```

3.2. Méthode dichotomique

a) Raisonnement par récurrence

Hypothèse: **V[1..inf-1].numéro<num<V[sup+1].numéro**



. inf=sup+1 alors V[1..inf-1].numéro<num<V[inf].numéro
donc **existe** := faux et **posit** := inf

. inf≤sup soit alors m := (inf+sup) div 2
..V[m].numéro=num alors **existe** := vrai et **posit** := m;
.. V[m].numéro<num alors inf := m+1 permet de retrouver l'hypothèse
.. V[m].numéro>num alors sup := m-1 permet de retrouver l'hypothèse

Initialisation: Pour vérifier l'hypothèse, il suffit de vérifier n>0 puis de poser inf := 1 et sup := n. Pour accélérer la recherche, on peut traiter à part les deux cas limites : num<V[1].numéro et num>V[n].numéro.

b) Algorithme

```
var  inf, sup, m : entier;
     existe : booléen;
début
     existe := faux;
     {initialisations}
     si n = 0 alors  posit := 1
     sinon
          si V[1].numéro>num alors
               posit := 1
          sinon
               si num >V[n].numéro alors
                    posit := n+1
               sinon
               début {V[1].numéro ≤ num  ≤ V[n].numéro}
                    inf := 1;  sup := n;
                    {V[1..inf-1].numéro<num<V[sup+1].numéro}
                    tantque (inf≤sup) et non existe faire
                    début
                         m := (inf+sup) div 2;
                         si V[m].numéro=num alors
                              existe := vrai;
                         sinon
                              si V[m].numéro<num alors
                                   inf := m+1
                              sinon
                                   sup := m-1
                    fin;
                    {(inf = sup  + 1) ou existe,
                              V[1..inf-1].numéro<num<V[sup+1].numéro}
                    si existe alors
                         posit := m
                    sinon
                         posit := inf
               fin
fin ;
```

4. Pour pouvoir insérer une nouvelle voiture dans le vecteur, il faut d'abord vérifier que le vecteur n'est pas "plein", et que le numéro d'immatriculation nouveau n'y figure pas déjà. Pour cela, on utilisera la procédure recherche.

procédure newauto (dr V : tabv; dr n : entier; d auto : voiture;
r new: booléen);
spécification {**n** ∈ [0..100] , **V** trié sur les **numéros** croissants}
=> {(**auto.numéro** ∉ **V[1..n]** , **auto** a été inséré dans **V**, **new**)
∨ (**auto.numéro** ∈ **V[1..n]** ,¬ **new**)}

```
var  présent : booléen;
     place, i : entier;
début
     new := faux;
     si n < 100 alors
     début  {il reste de la place}
          recherche (V, n, auto.numéro, présent, place);
          si  non  présent alors {voiture absente}
          début  {insertion possible ]
               new := vrai;
               pour i:= n bas place faire  V[i+1] := V[i];
               n := n+1;
               V[place] := auto
          fin
     fin
fin;
```

5. A l'inverse, pour supprimer un élément du vecteur, il faut vérifier que cet élément y existe bien.

```
procédure casse (dr V : tabv; dr n : entier; d num : chaîne8;  r old : booléen);
spécification {n ∈ [0..100], V trié sur les numéros croissants}
               => {num ∈ V[1..n] , la voiture  a été supprimée dans V, old)
                                                  ∨  (num ∉ V[1..n] , ¬ old)}
var  i, place : entier;
début
     recherche(V,n, num, old,place);
     {old = voiture présente , suppression possible}
     si old alors
     début
          n := n-1;
          pour i := place haut n faire  V[i] := V[i+1];
     fin
fin;
```

6. Il suffit ici de reprendre la procédure du cours (Tome 1, p. 160) en adaptant le test de comparaison de deux éléments consécutifs au(x) critère(s) souhaité(s).

```
procédure retrier (dr V: tabv; d n : entier);
spécification {n ∈ [0..100] , V trié sur les numéros croissants}
               => {V est trié sur les marques dans l'ordre alphabétique,
                         et à marque égale, sur les prix décroissants}
var  i, j : entier;
     perm : booléen;
```

```
début
    i := n;
    perm := vrai;
    tantque perm faire
    début
        perm := faux;
        pour j := 1 haut i-1 faire
        si  ((V[j].marque=V[j +1].marque) et (V[j].prix<V[j +1].prix))
                                ou (V[j].marque>V[j +1].marque) alors
        début
            permut (V, j, j +1);
            perm := vrai
        fin;
        i := i-1
    fin
fin;
```

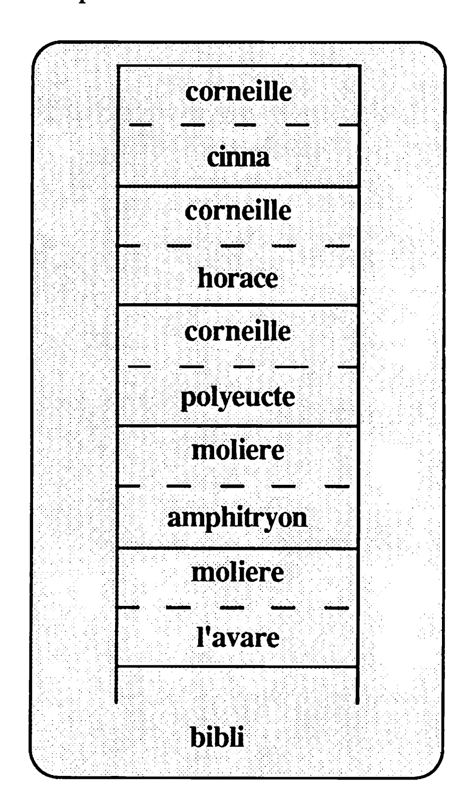
3.2. Catalogue d'une bibliothèque

Énoncé

On souhaite gérer les livres d'une bibliothèque à l'aide d'une structure composée d'un vecteur **bibli[1..nmax]**. Chaque élément du vecteur bibli est une variable structurée composée de deux champs : **nomaut** et **titre** permettant respectivement d'indiquer le nom de l'auteur et le titre du livre qu'il a écrit.

Le vecteur **bibli** est **trié par ordre alphabétique** des auteurs et pour chaque auteur **par ordre alphabétique sur les titres** écrits par chaque auteur.

Cette organisation correspond au schéma suivant :



On dispose des déclarations suivantes :
const
  nmax = 1000 ;

```
type
    ch20      =   chaîne20 ;
    livre     =   structure
                      nomaut , titre : ch20 ;
                  fin;
    vecteur   =   tableau [1..nmax] de livre;
```

On supposera que les titres de tous les livres d'un même auteur sont différents.
Dans toutes les questions, on supposera que **1 ≤ n < nmax**

1. PARCOURS DE VECTEURS

1.1. Écrire, sous forme itérative puis récursive, une
fonction nbaut (d bibli : vecteur; d n : entier) : entier ;
spécification {**n ≥ 1, bibli[1..n] trié**} =>
{**nbaut** = nombre d'auteurs présents dans **bibli[1..n]**}

1.2. Écrire, sous forme d'une recherche séquentielle puis d'une recherche dichotomique, une
fonction pointaut (d bibli : vecteur; d n : entier ; d nom : ch20) : entier;
spécification {**n ≥ 1, bibli[1..n] trié**} => {(**pointaut** = indice sur la **première occurrence** de l'auteur **nom** dans **bibli[1..n]**)
v (**pointaut** = **0** , l'auteur **nom** est absent}

1.3. Écrire une
fonction pointlivre (d bibli: vecteur; d n: entier; d nom , titre: ch20) : entier;
spécification {**n ≥ 1, bibli[1..n] trié**} => {(**pointlivre** = indice sur le livre **titre** écrit par l'auteur **nom** dans **bibli[1..n]**)
v (**pointlivre** = **0** ,le livre **titre** écrit par **nom** n'existe pas.)}

1.4. Écrire une
fonction aécrit (d bibli : vecteur; d n : entier ; d nom ,titre : ch20) : booléen;
spécification {**n ≥ 1, bibli[1..n] trié**} =>
{**aécrit** = l'auteur **nom** a écrit le livre **titre**}

1.5. Écrire une
fonction nblivraut (d bibli : vecteur; d n : entier ; d nom : ch20) : entier;
spécification {**n ≥ 1, bibli[1..n] trié**} => {**nblivraut** = nombre de livres écrits par l'auteur **nom** dans **bibli[1..n]**}

1.6. Écrire une
fonction nblivrautmax (d bibli : vecteur; d n : entier) : entier;
spécification {**n ≥ 1, bibli[1..n] trié**} => {**nblivrautmax** = indice de la **première occurrence** de l'auteur ayant écrit le plus de livres dans **bibli[1..n]**}

2. INSERTIONS D'ÉLÉMENTS

2.1. Écrire une
fonction placeinsert (d bibli :vecteur; d n : entier; d nom,titre :ch20) :entier;
spécification {**n ≥ 1, bibli[1..n] trié**} => {(**placeinsert** = place de l'insertion
de **titre** écrit par **nom** , **titre** ∉ **bibli[1..n]**)
∨ (**placeinsert** = indice du livre **titre** écrit par **nom**
, **titre** ∈ **bibli[1..n]**)}

2.2. Écrire une
procédure insertion (dr bibli: vecteur; dr n : entier; d nom,titre : ch20);
spécification {**n ≥ 1, bibli[1..n] trié**} =>
{**titre** écrit par **nom** a été inséré dans **bibli[1..n]**}

3. SUPPRESSIONS D'ÉLÉMENTS

On dispose maintenant d'un champ supplémentaire **présent** dans la structure livre :

```
livre =   structure
              nomaut , titre : ch20;
              présent : booléen
          fin ;
```

Ce booléen **présent** est utilisé pour effectuer une mise à jour logique du vecteur bibli. Sa valeur est à **faux** dans le cas où le livre a été perdu, sa valeur est à **vrai** dans le cas contraire.

3.1. Écrire une
procédure suplog (dr bibli: vecteur; d n : entier; d nom,titre: ch20);
spécification {**n ≥ 1, bibli[1..n] trié**} => {le livre **titre** écrit par **nom** a été
supprimé logiquement dans **bibli[1..n]**}

3.2. Écrire une
procédure sup (dr bibli: vecteur; dr n : entier);
spécification {**n ≥ 1, bibli[1..n] trié**} => {tous les livres perdus ont été
supprimés physiquement de **bibli[1..n]**}

Solutions proposées

1. PARCOURS DE VECTEURS

1.1. Pour ces deux fonctions, il s'agit d'adapter les algorithmes classiques de parcours de vecteurs (cf Tome 1, p. 128) .

Version itérative : première solution

```
fonction nbaut (d bibli : vecteur; d n : entier) : entier ;
spécification {n ≥ 1, bibli[1..n] trié} =>
                         {nbaut = nombre d'auteurs présents dans bibli[1..n]}
var i, nbt : entier;
début
    nbt := 1; i := 1;
    tantque i < n faire
    début
        si bibli[i].nomaut ≠ bibli[i + 1].nomaut alors
            nbt := 1 + nbt;
        i := i + 1
    fin;
    nbaut := nbt;
fin;
```

Version itérative : deuxième solution

On n'utilise plus de variable auxiliaire i, on se sert de n qui est protégée.

```
fonction nbaut (d bibli : vecteur; d n : entier) : entier ;
spécification {n ≥ 1, bibli[1..n] trié} =>
                         {nbaut = nombre d'auteurs présents dans bibli[1..n]}
var nbt : entier;
début
    nbt := 1;
    tantque n ≠ 1 faire
    début
        si bibli[n].nomaut ≠ bibli[n - 1].nomaut alors
            nbt := 1 + nbt;
        n := n - 1
    fin;
    nbaut := nbt;
fin;
```

Version récursive : C'est la traduction de la deuxième solution de la version itérative (cf tome 1, p. 132)

```
fonction nbaut (d bibli : vecteur; d n : entier) : entier ;
spécification {n ≥ 1, bibli[1..n] trié} => {nbaut = nombre d'auteurs
                                              présents dans bibli[1..n]}
```

```
début
    si n = 1 alors
        nbaut := 1
    sinon
        si bibli[n].nomaut ≠ bibli[n - 1].nomaut alors
            nbaut := 1 + nbaut(bibli, n - 1)
        sinon
            nbaut := nbaut(bibli, n - 1)
fin;
```

1.2. Recherche séquentielle : première version

C'est l'algorithme classique de recherche séquentielle dans un vecteur trié. On s'inspire de l'algorithme défini dans le Tome 1, p. 135.

```
fonction pointaut (d bibli : vecteur; d n : entier ; d nom : ch20) : entier;
spécification {n ≥ 1, bibli[1..n] trié} => {(pointaut = indice sur la
                        première occurrence de l'auteur nom dans bibli[1..n])
                                    v(pointaut = 0 , l'auteur nom est absent}
var i : entier;
début
    si (bibli[1].nomaut > nom) ou (bibli[n].nomaut < nom) alors
        pointaut := 0
    sinon
    début
        i := 1; {bibli[i].nomaut ≤ nom ≤ bibli[n].nomaut}
        tantque bibli[i].nomaut < nom faire
            {bibli[i].nomaut <  nom ≤ bibli[n].nomaut}
            i := i + 1;
        {(i = 1 , bibli[1].nomaut =  nom)
        ou (1 < i ≤ n , bibli[i-1].nomaut <  nom ≤ bibli[i].nomaut}
        si bibli[i].nomaut = nom alors  pointaut := i
        sinon  pointaut := 0
    fin
fin;
```

Recherche séquentielle : deuxième version

On utilise une variable auxiliaire booléenne **infer.**

```
fonction pointaut (d bibli : vecteur; d n : entier ; d nom : ch20) : entier;
spécification {n ≥ 1, bibli[1..n] trié} => {(pointaut = indice sur la
                        première occurrence de l'auteur nom dans bibli[1..n])
                                    v(pointaut = 0 , l'auteur nom est absent}
var i : entier;
    infer : booléen;
début
    i := 1;
    infer := vrai ;
    pointaut := 0 ;
```

```
    tantque (i ≤ n) et infer faire
        si bibli[i].nomaut < nom alors
            {bibli[i].nomaut < nom}
            i := i + 1 {bibli[i - 1].nomaut < nom}
        sinon {bibli[i].nomaut ≥ nom}
            infer := faux ;
    {(i = 1 , bibli[1].nomaut ≥ nom , non infer)
    v(1 < i ≤ n , bibli[i-1].nomaut < nom ≤ bibli[i].nomaut , non infer)
            v (i = n + 1 , bibli[n].nomaut < nom , infer)}
    si non infer alors
        si bibli[i].nomaut = nom alors pointaut := i
fin;
```

La première solution a notre préférence.

Recherche dichotomique : C'est l'algorithme classique de recherche dichotomique dans un vecteur trié. On s'inspire de l'algorithme défini dans le Tome 1 p. 141.

```
fonction pointaut (d bibli : vecteur; d n : entier ; d nom : ch20) : entier;
spécification {n ≥ 1, bibli[1..n] trié} => {(pointaut = indice sur la
                        première occurrence de l'auteur nom dans bibli[1..n])
                                    v(pointaut = 0 , l'auteur nom est absent}
var inf, sup, milieu : entier
début
    si nom > bibli[n].nomaut alors pointaut := 0
    sinon
    début
        inf := 1; sup := n;
        {bibli[inf-1].nomaut < nom ≤ bibli[sup].nomaut}
        tantque inf < sup faire
        début
            milieu := (inf + sup) div 2;
            si nom ≤ bibli[milieu].nomaut alors
                sup := milieu
            sinon inf := milieu + 1
            {bibli[inf-1].nomaut < nom ≤ bibli[sup].nomaut}
        fin;
        {bibli[inf-1].nomaut < nom ≤ bibli[inf].nomaut}
        si bibli[inf].nomaut = nom alors pointaut := inf
        sinon pointaut := 0
    fin;
fin;
```

1.3. Dans un premier temps, on recherche l'auteur ; si on l'a trouvé, on poursuit par une recherche séquentielle triée sur le livre écrit par l'auteur.

Première version :

```
fonction pointlivre (d bibli: vecteur; d n: entier; d nom , titre: ch20) : entier;
```

```
spécification {n ≥ 1, bibli[1..n] trié} => {(pointlivre = indice sur le livre
                                titre écrit par l'auteur nom dans bibli[1..n])
                  v(pointlivre = 0 ,le livre titre écrit par nom n'existe pas.)}
var i : entier;
début
    i := pointaut(bibli, n, nom); {recherche de l'auteur}
    si i = 0 alors {l'auteur n'existe pas} pointlivre := 0
    sinon
    début {recherche séquentielle triée du livre écrit par l'auteur}
        si (bibli[n].nomaut = nom) et (bibli[n].titre < titre) alors
            pointlivre := 0
        sinon
        début
            tantque (bibli[i].nomaut = nom) et (bibli[i].titre < titre) faire
                i := i + 1;
            si (bibli[i].nomaut = nom) et (bibli[i].titre = titre) alors
                pointlivre := i
            sinon pointlivre := 0
        fin;
    fin;
fin;
```

Deuxième version : on utilise une variable booléenne **infer**.

```
fonction pointlivre (d bibli: vecteur; d n: entier; d nom , titre: ch20) : entier;
spécification {n ≥ 1, bibli[1..n] trié} => {(pointlivre = indice sur le livre
                                titre écrit par l'auteur nom dans bibli[1..n])
                  v(pointlivre = 0 ,le livre titre écrit par nom n'existe pas.)}
var i : entier; infer: booléen ;
début
    i := pointaut(bibli, n, nom); {recherche de l'auteur}
    pointlivre := 0 ;
    si i ≠ 0 alors
    début {recherche séquentielle triée du livre écrit par l'auteur}
        infer := vrai ;
        tantque (i ≤ n) et infer faire
            si (bibli[i].nomaut = nom) et (bibli[i].titre < titre) alors
                i := i + 1
            sinon infer := faux ;
        {(i ≤ n ,(bibli[i].nomaut ≠ nom) v(bibli[i].titre ≥ titre) ,non infer)
        v(i = n + 1, bibli[n].nomaut = nom , bibli[n].titre < titre , infer)}
        si non infer alors
            si (bibli[i].nomaut = nom) et (bibli[i].titre = titre) alors
                pointlivre := i
   fin;
fin;
```

La première version a encore notre préférence.

1.4. On utilise tout simplement la fonction **pointlivre** pour savoir si l'auteur **nom** a écrit le livre **titre** .

```
fonction aécrit (d bibli : vecteur; d n : entier ; d nom ,titre : ch20) : booléen;
spécification {n ≥ 1, bibli[1..n] trié} =>
                              {aécrit = l'auteur nom a écrit le livre titre}
début
    aécrit := pointlivre (bibli, n, nom, titre) ≠ 0
fin;
```

1.5. On procède en deux temps : recherche de l'auteur suivie du comptage des livres écrits par l'auteur (fonction **nblivre**).

```
fonction nblivraut (d bibli : vecteur; d n : entier ; d nom : ch20) : entier;
spécification {n ≥ 1, bibli[1..n] trié} => {nblivraut = nombre de livres
                              écrits par l'auteur nom dans bibli[1..n]}
var indaut : entier;
début
    indaut := pointaut (bibli, n, nom);
    {(indaut = indice de la première occurrence de nom dans bibli[1..n]
                , nom est présent) v(indaut = 0 , nom est absent}}
    si indaut = 0 alors
        nblivraut := 0
    sinon
        nblivraut := nblivre(bibli, n, indaut, nom)
fin;

fonction nblivre (d bibli : vecteur;d n , indaut : entier;d nom : ch20) : entier;
spécification {0 < indaut ≤ n , bibli[1..n] trié} => {nblivre = nombre
                d'occurrences de nom à partir de indaut dans bibli[1..n]}
var nb : entier;
début
    si bibli[n].nomaut = nom alors {nom est le dernier élément de bibli}
        nblivre := n - indaut + 1
    sinon
    début {bibli[n].nomaut > nom}
        nb := 1;
        indaut := indaut + 1;
        tantque bibli[indaut].nomaut = nom faire
        début
            nb := nb + 1;
            indaut := indaut + 1
        fin;
        nblivre := nb
    fin
fin;
```

1.6. Il s'agit d'un parcours classique de vecteur ; pour chaque auteur, on calcule le nombre de livres écrits en gardant dans la variable **autmax** l'indice sur l'auteur ayant écrit le plus de livres : **max.**

```
fonction nblivrautmax (d bibli : vecteur; d n : entier) : entier;
spécification {n ≥ 1, bibli[1..n] trié} => {nblivrautmax = indice de la
                        première occurrence de l'auteur ayant écrit le plus
                                                de livres dans bibli[1..n]}
var nb, max, indaut, autmax : entier;
début
    indaut := 1; autmax := 1;
    max := 0;
    tantque indaut ≤ n faire
    début
        nb := nblivre (bibli, n, indaut, bibli[indaut].nomaut);
        si max < nb alors
        début
            max := nb;
            autmax := indaut
        fin;
        indaut := indaut + nb
    fin;
    nblivrautmax := autmax
fin;
```

2. INSERTIONS D'ÉLÉMENTS

2.1. La recherche se fait en deux temps : recherche par dichotomie de l'auteur suivie de la recherche séquentielle triée du livre.

```
fonction placeinsert (d bibli :vecteur; d n : entier; d nom,titre :ch20) :entier;
spécification {n ≥ 1, bibli[1..n] trié} => {(placeinsert = place de l'insertion
                              de titre écrit par nom , titre ∉ bibli[1..n])
                        v(placeinsert = indice du livre titre écrit par nom
                                                      , titre ∈ bibli[1..n])}

var i, inf, sup, milieu : entier;
début
    si (bibli[1].nomaut ≥ nom) alors
        i := 1
    sinon
        si (bibli[n].nomaut < nom)
          ou ((bibli[n].nomaut = nom) et (bibli[n].titre < titre)) alors
            i := n + 1
        sinon
            {(nom < bibi[1].nomaut) v (bibli[n].nomaut > nom)
                    v ((bibli[n].nomaut = nom),(bibli[n].titre ≥ titre))}
```

```
        début {recherche de l'auteur}
            inf := 1; sup := n;
            {bibli[inf].nomaut <  nom ≤ bibli[sup].nomaut}
            tantque inf ≠ sup - 1 faire
            début
                milieu := (inf + sup) div 2;
                si bibli[milieu].nomaut < nom alors
                    inf := milieu
                sinon
                    sup := milieu
                {bibli[inf].nomaut <  nom ≤ bibli[sup].nomaut}
            fin;
            {bibli[sup - 1].nomaut <  nom ≤ bibli[sup].nomaut}
            i := sup;
        fin;
    {recherche du livre}
    tantque (bibli[i].nomaut = nom) et (bibli[i].titre < titre) faire
        i := i + 1;
    placeinsert := i
fin;
```

2.2. Après la recherche de la place de l'insertion, on effectue les décalages nécessaires avant de réaliser l'insertion de l'élément.

```
procédure insertion (dr bibli: vecteur; dr n : entier; d nom,titre : ch20);
spécification {n ≥ 1, bibli[1..n] trié} =>
                        {titre écrit par nom a été inséré dans bibli[1..n]}
var  place, i : entier;  inser : booléen;
début
    {recherche de la place}
    place := placeinsert (bibli, n, nom, titre);
    inser := faux ;  {critère d'insertion}
    si place = n + 1 alors
        inser := vrai
    sinon  {1 ≤place  ≤ n}
        si (bibli[place].nomaut ≠ nom) ou (bibli[place].titre ≠ titre) alors
            inser := vrai;

    si inser alors
    début
        n := n + 1;
        i := n;
        tantque i > place faire
        début  {décalage des éléments du vecteur}
            bibli[i] := bibli[i - 1];
            i := i - 1
        fin;
```

```
        {insertion du nouvel élément}
        bibli[place].nomaut := nom;
        bibli[place].titre := titre;
    fin;
fin;
```

3. SUPPRESSIONS D'ÉLÉMENTS

3.1.

```
procédure suplog (dr bibli: vecteur; d n : entier; d nom,titre: ch20);
spécification {n ≥ 1, bibli[1..n] trié} => {le livre titre écrit par nom a été
                                  supprimé logiquement dans bibli[1..n]}
var  plivre : entier;
début
    plivre := pointlivre(bibli, n, nom , titre);
    si plivre ≠ 0 alors
        bibli[plivre].présent := faux;
fin;
```

3.2.

Il s'agit ici de l'adaptation de l'algorithme défini dans le Tome 1, p. 182.

```
procédure sup (dr bibli: vecteur; dr n : entier);
spécification {n ≥ 1, bibli[1..n] trié} => {tous les livres perdus ont été
                                  supprimés physiquement de bibli[1..n]}
var i : entier;
début
    {recherche de l'indice i du premier élément à supprimer}
    i := premiersup(bibli, n);
    retasser(bibli, n, i)
fin;
```

```
fonction premiersup (d bibli : vecteur;  d n : entier) : entier;
spécification {n ≥ 1, bibli[1..n] trié} =>
              {premiersup = indice du premier élément à supprimer}
var i : entier;
début
    i := 1;
    tantque (bibli[i].présent) et (i < n) faire
        i := i + 1;
    si non (bibli[i].présent) alors
        premiersup := i
    sinon {pas d'éléments à supprimer}
        premiersup := n + 1
fin;
```

```
procédure retasser (dr bibli : vecteur;  dr n : entier;  d i : entier);
spécification {n ≥ 1, bibli[1..n] trié} =>
                    {les éléments de bibli[1..n] ont été retassés à partir de i}
var j : entier;
début
    j := i + 1;
    tantque j ≤ n faire
    début
        si bibli[j].présent alors
        début
            bibli[i] := bibli[j];
            i := i + 1
        fin;
        j := j + 1
    fin;
    n := i - 1
fin;
```

3.3. Course de ski

Énoncé

On souhaite gérer la saisie et l'affichage des résultats d'une course de ski. Pour cela, on dispose, avant la course, d'un fichier séquentiel contenant le nom et la nationalité de chacun des concurrents inscrits. Les dossards sont attribués aux concurrents dans l'ordre où ils figurent dans le fichier. On supposera que ce fichier est chargé en mémoire dans un vecteur, de telle sorte que le numéro de dossard d'un concurrent soit égal à son indice dans le vecteur.

Au moment de la course, les concurrents prennent le départ les uns après les autres, à une minute d'intervalle, dans un ordre qui n'est pas nécessairement celui des dossards. A l'arrivée de chaque concurrent, on saisit directement son numéro de dossard et son temps (en minutes, secondes et centièmes). On veut obtenir alors l'affichage du classement provisoire de la course : dans l'ordre croissant des temps de parcours, on affichera le rang, le nom et le temps, de chaque concurrent déjà classé. Ce classement s'allongera au fur et à mesure du déroulement de la course.

Exemple : soit le fichier de skieurs suivant :

Pellen	*France*
Courtin	*France*
Stenmark	*Suède*
Goitschel	*France*
Killy	*France*
Kowarski	*France*
Tomba	*Italie*

Premier résultat saisi :

4 2 45 78

Premier affichage :

1 Goitschel 2 45 78

Deuxième résultat saisi:

7	5 56 89

Deuxième affichage :

1	Goitschel	2 45 78
2	Tomba	5 56 89

Troisième résultat saisi :

1	2 45 78

Troisième affichage :

1	Goitschel	2 45 78
2	Pellen	2 45 78
3	Tomba	5 56 89

On utilisera les déclarations de types suivants :

```
type ch10    =  chaîne10;
     ttemps  =  structure
                    min, sec : 0..59;
                    cent : 0..99
                fin;
     tskieur =  structure
                    nom, nationalité : ch10
                fin;
     tclass  =  structure
                    nom :ch10;
                    temps :ttemps
                fin;
     tvski   =  tableau [ 1..100] de tskieur;
     tvcl    =  tableau [ 1..100] de tclass;
```

On supposera qu'il existe une procédure qui permet la saisie du numéro de dossard et du temps, d'en-tête :
procédure saisie (r doss:entier; r temps : ttemps);

1. FONCTIONS ET PROCÉDURES DE BASE

1.1. Écrire la fonction suivante, qui permet la comparaison de deux temps :
fonction infeg (d t1, t2 : ttemps) : booléen ;
spécification { } => {**infeg = t1** est inférieur ou égal à **t2**}

1.2. Écrire la fonction suivante, qui permet de trouver la position de **temps** parmi les temps déjà enregistrés dans vcl [1..nb] :

fonction posit (d vcl : tvcl ; d nb : entier; d temps : ttemps) : entier ;
spécification {**nb >0, vcl [1..nb]** trié} =>
{**vcl [1..posit - 1] .temps ≤ temps < vcl [posit .. nb] .temps ,**
posit ∈ [1..nb+1]}

1.3. Écrire la procédure suivante, qui permet d'insérer le nom correspondant à **doss** et le **temps** donné, à la **place** donnée, dans le vecteur trié **vcl [1..nb]** . Le nom sera extrait du vecteur **vski.**
procédure insertplace (d place, doss : entier; d vski : tvski; d temps: ttemps;
dr vcl : tvcl; dr nb : entier);

1.4. Écrire la procédure suivante, qui permet l'insertion dans **vcl** du nom correspondant à **doss** et du **temps** donné, de telle sorte que vcl reste trié. La variable **nb** donne le nombre courant d'éléments de vcl.
procédure insert (d doss : entier; d vski : tvski; d temps : ttemps;
dr vcl : tvcl ; dr nb : entier);

1.5. Écrire la procédure suivante, qui permet l'affichage du classement des **nb** premiers concurrents :
procédure affichclass (d vcl : tvcl; d nb : entier);

1.6. Écrire la procédure qui permet d'enchaîner les opérations de saisie et d'affichage des classements, puis délivre le classement final et le nombre total de concurrents dans **vcl [1..nb]**. On supposera que la fin de la saisie est indiquée par un numéro de dossard égal à 0.
procédure course (d vski : tvski; r vcl : tvcl ; r nb : entier) ;

2. TRAITEMENT DES EX AEQUO

On veut maintenant perfectionner l'affichage, en attribuant un rang égal à deux concurrents arrivés dans le même temps ("ex aequo").
Exemple : le troisième classement deviendrait :

1	Goitschel	2 45 78
1	Pellen	2 45 78
3	Tomba	5 56 89

Modifier la procédure **affichclass** à cet effet.

3. TRAITEMENT DES DISQUALIFIES

On suppose maintenant que certains concurrents peuvent avoir manqué le passage d'une porte pendant leur course, ils sont alors disqualifiés. La procédure de saisie est améliorée ; elle délivre, en plus du numéro de dossard et du temps, un booléen "qual" qui n'est vrai que si le concurrent n'est pas disqualifié :

procédure saisie2 (r doss:entier; r temps : ttemps; r qual : booléen);

On va donc pouvoir traiter à part les concurrents disqualifiés, en se contentant de construire une liste de leurs noms et nationalités, dans l'ordre des saisies. On utilisera pour cela un vecteur **vdis**, de type tvski.

3.1. Écrire la procédure qui permet l'ajout du concurrent de dossard **doss** à la fin de **vdis [1..nbdis]**.

procédure inserdis (dr vdis : tvski; dr nbdis : entier; d doss : entier; d vski : tvski) ;

3.2. Une fois la liste des concurrents disqualifiés construite, dans l'ordre des départs, on souhaite l'obtenir dans l'ordre alphabétique des noms. Pour cela, on va trier la liste au moyen d'un "tri par segmentation".

Écrire les algorithmes nécessaires pour obtenir ce tri, appelé par l'instruction : **triseg (vdis, 1, nbdis).**

procédure triseg (dr v : tvski; d inf,sup : entier);

3.3. Écrire la procédure qui permet d'afficher la liste complète des noms et nationalités des concurrents disqualifiés :

procédure affichdisqual (d vdis : tvski; d nbdis : entier);

3.4. Modifier la procédure **course** écrite ci-dessus, pour enchaîner les opérations de saisie, d'affichage des classements successifs, et enfin d'affichage de la liste des disqualifiés :

procédure coursebis (r vcl : tvcl; r vdis : tvski; r nb, nbdis : entier);

Solutions proposées

1. FONCTIONS ET PROCÉDURES DE BASE

1.1. Cette fonction est très proche de la comparaison de deux dates, traitée dans le premier exercice de ce tome.

fonction infeg (d t1, t2 : ttemps) : booléen ;

spécification { } => {**infeg** = **t1** est inférieur ou égal à **t2**}

```
début
    infeg := (t1.min < t2.min)
             ou
             ((t1.min = t2.min) et (t1.sec < t2.sec))
             ou
             ((t1.min = t2.min) et (t1.sec = t2.sec) et (t1.cent ≤ t2.cent))
fin;
```

1.2. Cette fonction est une application directe de l'algorithme du cours (Tome 1, p. 173). On y remplace simplement les comparaisons directes par des appels à la fonction **infeg**.

```
fonction posit (d vcl : tvcl ; d nb : entier; d temps : ttemps) : entier ;
spécification {nb >0, vcl [ 1..nb] trié} =>
             {vcl [ 1..posit - 1 ] .temps ≤ temps < vcl [ posit .. nb ] .temps ,
                                                          posit ∈ [1..nb+1]}
var i:entier;
début
    si infeg (vcl [ nb].temps, temps) alors
        posit := nb+1
    sinon
    début
        i := 1;
        tantque infeg (vcl [ i].temps,temps) faire
            i := i+1;
        posit := i
    fin
fin;
```

1.3. Il ne faut pas oublier de "décaler" d'une position vers le bas tous les éléments d'indice supérieur ou égal à **place**, en commençant par le bas, ni d'incrémenter **nb**. On veut insérer dans le classement le nom du skieur plutôt que son numéro de dossard, on utilise donc un accès direct au vecteur vski.

```
procédure insertplace (d place, doss : entier; d vski : tvski; d temps: ttemps;
                                              dr vcl : tvcl; dr nb : entier);
var i:entier;
début
    pour i := nb bas place faire
        vcl [ i+1] := vcl [ i];
    nb := nb+1;
    vcl [ place].nom := vski [ doss].nom;
    vcl [ place].temps := temps
fin;
```

1.4. Il s'agit encore d'une application directe de l'algorithme du cours (Tome 1, p. 171). Bien entendu, on utilise ici tous les algorithmes demandés ci-dessus.

```
procédure insert (d doss : entier; d vski : tvski; d temps : ttemps;
                                          dr vcl : tvcl ; dr nb : entier);
var place : entier;
début
    si nb=0 alors
    début
        vcl [ 1].nom := vski [ doss].nom;
        vcl [ 1].temps := temps;
        nb := 1
    fin
    sinon
    début
        place := posit (vcl,nb,temps) ;
        insertplace (place, doss, vski, temps, vcl, nb) ;
    fin
fin;
```

1.5. Cette procédure est un simple affichage de chacun des **nb** éléments de vcl, précédé de son indice dans vcl.

```
procédure affichclass (d vcl : tvcl; d nb : entier);
var   i : entier;
début
    pour i := 1 haut nb faire
        écrireln (i, vcl [ i].nom,
                    vcl [ i].temps.min, vcl [ i].temps.sec, vcl [ i].temps.cent);
fin;
```

1.6. Il suffit d'initialiser le nombre de skieurs à 0, puis de répéter, au moyen d'une boucle itérative, les opérations de saisie, d'insertion des valeurs saisies dans vcl, et d'affichage, en s'arrêtant si le dossard saisi est nul. La première saisie se fera avant l'entrée dans la boucle.

```
procédure course (d vski : tvski; r vcl : tvcl ; r nb : entier) ;
var   doss : entier; temps : ttemps;
début
    nb := 0;
    saisie(doss, temps);
    tantque doss ≠ 0 faire
    début
        insert (doss, vski, temps, vcl, nb);
        affichclass (vcl, nb);
        saisie (doss, temps)
    fin;
fin ;
```

2. TRAITEMENT DES EX AEQUO

Il faut comparer le temps du nouveau concurrent à celui qui le précède. S'ils sont égaux, on garde le rang, sinon le rang reprend la valeur de l'indice.

```
procédure affichclass2 (d vcl : tvcl; d nb:entier);
var i,r:entier;
début
    r := 1;
    pour i := 1 haut nb faire
    début
        si non infeg (vcl [ i].temps, vcl [ r].temps) ou (i=1) alors r := i;
        écrireln(r, vcl [ i].nom,
                vcl [ i].temps.min, vcl [ i].temps.sec, vcl [ i].temps.cent)
    fin;
fin;
```

3. TRAITEMENT DES DISQUALIFIES

3.1. Insertion d'un nouvel élément à la fin d'un vecteur (Tome 1, p. 170).

```
procédure inserdis (dr vdis : tvski; dr nbdis : entier; d doss : entier;
                                                        d vski : tvski) ;
début
  nbdis := nbdis+1;
  vdis [ nbdis] := vski [ doss]
fin;
```

3.2. Il suffit de reprendre le tri du Tome 1, p. 187, avec la deuxième version de la procédure **segmentation**.

```
procédure permut (dr v : tvski ; d i, j : entier) ;
var sauve : tskieur ;
début
    sauve := v [ i ];
    v [ i ] := v [ j ];
    v [ j ] := sauve;
fin;

procédure segment  (dr  v : tvski;  d inf, sup : entier;  r  place : entier);
var   i,j : entier;
      nom:ch10;
début
    nom := v [ inf].nom;
    i := inf+1;
    j := sup;
```

```
        tantque i≤j faire
        début
            si v [ i].nom ≤ nom alors i := i+1
            sinon
            début
                tantque v [ j] .nom > nom faire
                    j := j-1;
                si i<j alors
                début
                    permut (v, i, j) ;
                    i := i+1;
                    j := j-1
                fin
            fin
        fin;
        permut (v, inf, j) ;
        place := j
    fin;

    procédure triseg (dr v : tvski; d inf, sup : entier);
    var place:entier;
    début
        si inf < sup alors
        début
            segment (v, inf, sup, place);
            triseg (v, inf, place-1);
            triseg (v, place+1, sup)
        fin
    fin;
```

3.3. Parcours évident :

```
procédure affichdisqual (d vdis : tvski;  d nbdis : entier);
var i:entier;
début
    écrireln ('disqualifiés:');
    pour i := 1 to nbdis faire
        écrireln (vdis [ i].nom, vdis [ i].nationalité);
fin;
```

3.4. On ajoute à la première version : l'initialisation du nombre de disqualifiés, le traitement de ceux-ci pendant la course, et enfin le tri et l'affichage des disqualifiés après la course.

```
procédure coursebis (r vcl : tvcl; r vdis : tvski; r nb,  nbdis : entier);
var   doss : entier;
      temps : ttemps;
      qual : booléen;
```

```
début
    saisie2 (doss, temps, qual);
    nb := 0;
    nbdis := 0;
    tantque doss ≠ 0 faire
    début
        si non qual alors
            inserdis (vdis, nbdis, doss, vski)
        sinon
        début
            insert (doss, vski, temps, vcl, nb);
            affichclass (vcl, nb)
        fin;
        saisie2 (doss, temps, qual)
    fin;
    triseg (vdis, 1, nbdis);
    affichdisqual (vdis, nbdis);
fin;
```

4

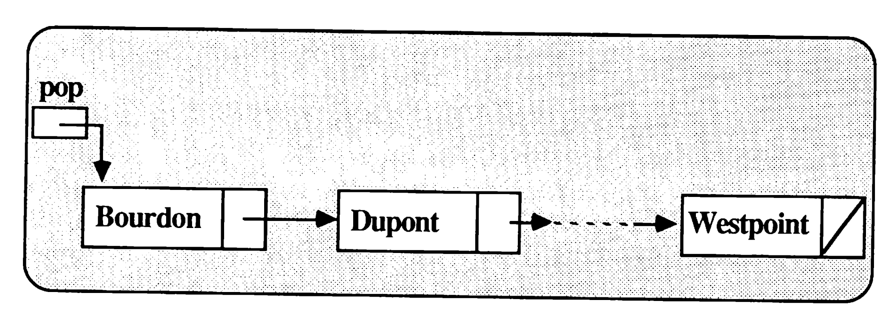
LISTES LINÉAIRES CHAINÉES

4.1. Familles

Énonce

1. PREMIERE PARTIE : LISTE À UN SEUL NIVEAU

On souhaite gérer une population composée de plusieurs familles limitées à une seule génération. Dans cette première partie, on se limitera à des familles composées uniquement du nom de familles. Dans la deuxième partie, on ajoutera les parents, les enfants et les voitures possédées par une famille. La liste est **ordonnée par ordre alphabétique** sur les **noms** de famille. La population correspond au schéma suivant :



On dispose des déclarations suivantes :

```
type
    ch20 =    chaîne20;
    pf   =    ↑famille;
    famille = structure
                  nom :     ch20;
                  suivant : pf;
              fin;
```

N.B. On appellera **adresse** de la famille **nom, l'adresse** de la cellule contenant le **nom** de famille.

1.1. ***Parcours de listes et comptage d'éléments***

1.1.1. Écrire, sous forme itérative puis récursive, une
fonction nbfamille (d pop : pf) : entier;
spécification { } => {**nbfamille** = nombre de familles ∈ **pop+**}

1.1.2. Écrire, sous forme itérative puis récursive, une
fonction présent (d pop : pf; d nomfam : ch20) : pf;
spécification {**pop+ triée**} =>{(**nomfam** ∈ **pop+**, **présent** = adresse de la famille **nomfam**) ∨ (**nomfam** ∉ **pop+**, **présent** = **nil**)}

1.2. Mise à jour d'éléments

1.2.1. Écrire, sous forme itérative, une
procédure inser (dr pop : pf; d nomfam : ch20);
spécification {**pop+ triée**} => {(**nomfam** ∉ **pop+**, **nomfam** a été inséré dans **pop+**) ∨ (**nomfam** ∈ **pop+**,pas d'insertion)}

1.2.2. Écrire la procédure **inser** sous forme récursive.

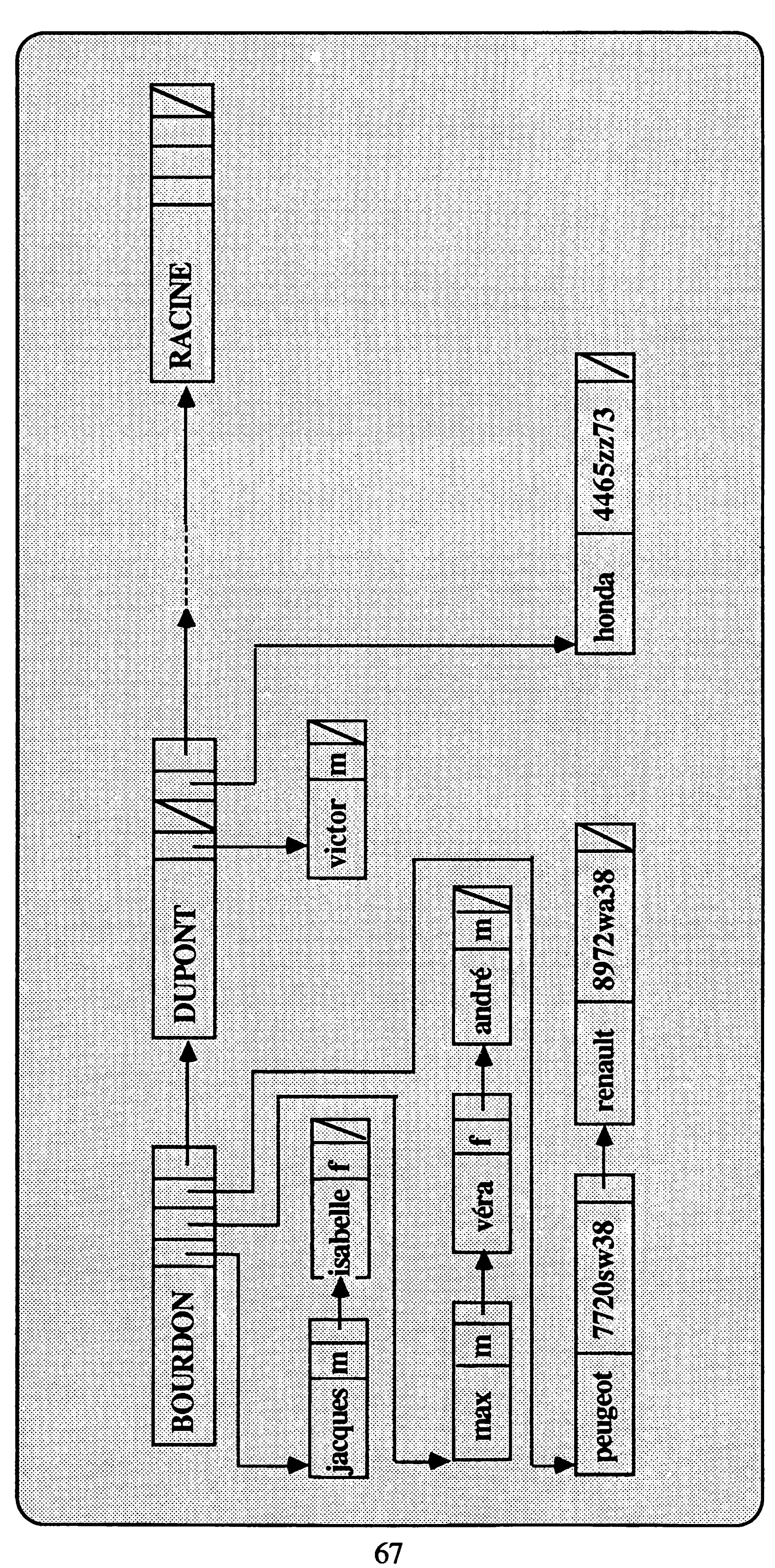
1.2.3. Écrire, sous forme itérative, une
procédure supp (dr pop : pf; d nomfam : ch20);
spécification {**pop+ triée**} => {(**nomfam** ∈ **pop+**, **nomfam** a été supprimé de **pop+**) ∨ (**nomfam** ∉ **pop+**, pas de suppression)}

1.2.4. Écrire la procédure **supp** sous forme récursive.

2. DEUXIEME PARTIE : LISTE À PLUSIEURS NIVEAUX

On considère une population composée de plusieurs familles. En général, une famille est composée de deux parents et d'un ou plusieurs enfants. On peut avoir aussi les cas particuliers suivants :
- un seul parent (pas de conjoint et pas d'enfant),
- un seul parent et un ou plusieurs enfants,
- aucun parent, mais un ou plusieurs enfants (cas du décès des deux parents),
- deux parents, mais pas d'enfant.

Une personne (parent ou enfant) appartient à une seule famille.
A chaque famille, on associe la liste des voitures, en nombre illimité, possédées par les membres de la famille.
On représentera la population par la figure de la page suivante.
On supposera que :
- toutes les familles ont un nom différent,
- tous les prénoms d'une même famille sont différents,
- le nombre de parents est limité à deux,
- le nombre des enfants est illimité,
- la population est **ordonnée par ordre alphabétique** sur les **noms** de famille.


BOURDON
DUPONT
RACINE
jacques
m
isabelle
f
victor
m
max
m
véra
f
andré
m
peugeot
7720sw38
renault
8972wa38
honda
4465zz73

On dispose alors des déclarations suivantes :

```
type
    ch20 =    chaîne20;
    pf   =    ↑famille;
    pp   =    ↑personne;
    pv   =    ↑voiture;

    famille = structure
                  nom :      ch20;
                  parent :   pp;
                  enfant :   pp;
                  voiture :  pv;
                  suivant :  pf;
              fin;
    personne =    structure
                  prénom :   ch20;
                  sexe :     car;
                  suivant :  pp;
              fin;
    voiture = structure
                  marque :   ch20;
                  numéro :   ch20;
                  suivant :  pv;
              fin;
```

2.1. Parcours de listes et comptage d'éléments

2.1.1. Écrire, sous forme itérative puis récursive, une
fonction nbfamillesansparent (d pop : pf) : entier;
spécification { } => {**nbfamillesansparent** = nombre de familles de **pop**$^+$ dont les parents sont décédés}

2.1.2. Écrire, sous forme récursive, une
fonction nbcélibat (d pop : pf) : entier;
spécification { } => {**nbcélibat** = nombre de familles de **pop**$^+$ n'ayant plus qu'un seul parent}

2.1.3. Écrire, sous forme itérative puis récursive, une
fonction nbenfant (d pop : pf) : entier;
spécification { } => {**nbenfant** = nombre d'enfants de **pop**$^+$}

2.1.4. Écrire, sous forme itérative puis récursive, une
fonction nbenfnomfam (d pop : pf; d nomfam : ch20) : entier;
spécification {**pop**$^+$ **triée**} =>{(**nomfam** $\in$ **pop**$^+$,**nbenfnomfam** = nombre d'enfants de la famille **nomfam**) $\vee$ (**nomfam** $\notin$ **pop**$^+$, **nbenfnomfam** = **0**)}

2.1.5. Un vol de voiture s'est produit. On désire connaître le propriétaire de la voiture. Écrire, sous forme récursive, une
fonction apparvoit (d pop : pf; d numéro : ch20) : pf;
spécification { } => {(voiture **numéro** ∈ **pop+,apparvoit** = adresse de la famille possédant cette voiture)∨ (voiture **numéro** ∉ **pop+,apparvoit = nil**)}

2.2. Mises à jour d'éléments

2.2.1. Écrire une
procédure naissance (d pop : pf; d nomfam, prénom : ch20; d sexe : car);
spécification {**pop+ triée**} => {(**nomfam** ∈ **pop+,prénom nomfam** du sexe **sexe** a été inséré dans **pop+**) ∨ (**nomfam** ∉ **pop+**,pas d'insertion)}

2.2.2. On suppose maintenant que la liste des enfants est ordonnée par ordre alphabétique. Écrire une
procédure naissancetrié (d pop : pf; d nomfam, prénom : ch20; d sexe : car);
spécification {**pop+ triée , liste des enfants de nomfam ordonnée**} => {(**nomfam** ∈ **pop+,prénom nomfam** du sexe **sexe** a été inséré dans **pop+**) ∨ (**nomfam** ∉ **pop+**,pas d'insertion)}

2.2.3. Il s'agit ici de supprimer une voiture. Écrire une
fonction suppvoiture (d pop : pf; d numéro : ch20) : booléen;
spécification { } => {(voiture **numéro** ∈ **pop+,suppvoiture** ,voiture **numéro a été supprimée**) ∨ (voiture **numéro** ∉ **pop+,¬suppvoiture**)}

2.2.4. Une famille achète une voiture à une autre famille. Écrire une
procédure achatvoit (d pop : pf ; d nom1, nom2, numéro : ch20);
spécification {**pop+ triée**} => {voiture **numéro** ∈ **famille nom2, nom1** ∈ **pop+,nom1** a acheté la voiture **numéro** de **nom2**)}

Solutions proposées

1. PREMIERE PARTIE : LISTE À UN SEUL NIVEAU

1.1. *Parcours de listes et comptage d'éléments*

1.1.1. Version itérative : parcours classique d'une liste (cf Tome 2, p. 23).
fonction nbfamille (d pop : pf) : entier;
spécification { } => {**nbfamille** = nombre de familles ∈ **pop+**}
var nb : entier;

```
début
    nb := 0;
    tantque pop ≠ nil faire
    début
        nb := nb + 1;
        pop := pop↑.suivant
    fin;
    nbfamille := nb
fin;
```

Version récursive : parcours d'une liste (cf Tome 2, p. 24).

```
fonction nbfamille (d pop : pf) : entier;
spécification { } => {nbfamille = nombre de familles ∈ pop+}
début
    si pop = nil alors
        nbfamille := 0
    sinon
        nbfamille := 1 + nbfamille(pop↑.suivant)
fin;
```

1.1.2.

Version itérative :

```
fonction présent (d pop : pf; d nomfam : ch20) : pf;
spécification {pop+ triée} =>{(nomfam ∈ pop+, présent = adresse de la
                        famille nomfam) v (nomfam ∉ pop+, présent = nil)}
var  sup : booléen;
début
    sup := vrai;
    présent := nil; {nomfam ∉ pop+, présent = nil}
    tantque (pop ≠ nil) et sup faire
        si pop↑.nom ≥ nomfam alors
            sup := faux
        sinon
            pop := pop↑.suivant;
    {(pop = nil) v (¬sup)}
    si pop ≠ nil alors {¬sup}
        si pop↑.nom = nomfam alors {nomfam ∈ pop+}
            présent := pop;
fin;
```

Version récursive : (cf Tome 2, p. 35)

```
fonction présent (d pop : pf; d nomfam : ch20) : pf;
spécification {pop+ triée} =>{(nomfam ∈ pop+, présent = adresse de la
                        famille nomfam) v (nomfam ∉ pop+, présent = nil)}
```

```
début
    si pop = nil alors présent := nil
    sinon  si pop↑.nom = nomfam alors présent := pop
        sinon  si pop↑.nom > nomfam alors présent := nil
            sinon {pop↑.nom < nomfam}
                présent := présent(pop↑.suivant, nomfam)
fin;
```

1.2. Mise à jour d'éléments

1.2.1.

Première version itérative : il s'agit d'une insertion classique dans une liste triée (cf Tome 2, p. 64). On définit d'abord la procédure auxiliaire **insertête** (cf Tome 2, p. 46).

```
procédure insertête (dr pfam : pf; d nomfam : ch20);
spécification { } => {insertion de nomfam en tête de pfam+}
var p : pf;
début
    nouveau(p);
    p↑.nom := nomfam; p↑.suivant := pfam;
    pfam := p;
fin;
```

```
procédure inser (dr pop : pf; d nomfam : ch20);
spécification {pop+ triée} => {(nomfam ∉ pop+, nomfam a été inséré dans
                                    pop+) v (nomfam ∈ pop+,pas d'insertion)}
var p, précèd : pf; égal, super : booléen;
début
    p := pop; {protection du paramètre pop passé en donnée-résultat}
    super := vrai;  égal := faux;
    tantque (p ≠ nil) et super faire
        si nomfam > p↑.nom alors
        début
            précèd := p;
            p := p↑.suivant;
        fin
        sinon
        début
            super := faux;
            égal := nomfam = p↑.nom
        fin;
    si non égal alors
        si p = pop alors {insertion en tête de liste}
            insertête(pop, nomfam)
        sinon
            insertête(précèd↑.suivant, nomfam);
fin;
```

Deuxième version itérative : on utilise le principe de la sentinelle consistant à ajouter en tête de la liste une cellule bidon évitant ainsi le traitement du cas particulier de l'insertion en tête.

```
procédure inser (d pop : pf; d nomfam : ch20);
spécification {pop+ triée} => {(nomfam ∉ pop+, nomfam a été inséré dans
                                pop+) v (nomfam ∈ pop+,pas d'insertion)}
var égal, super : booléen;
début
    super := vrai; égal := faux;
    {le premier élément significatif est en pop↑.suivant}
    tantque (pop↑.suivant ≠ nil) et super faire
        si nomfam > pop↑.suivant↑.nom alors pop := pop↑.suivant
        sinon
        début
            super := faux;
            égal := nomfam = pop↑.suivant↑.nom
        fin;
    si non égal alors insertête(pop↑.suivant, nomfam);
fin;
```

On remarquera que l'algorithme est simplifié et que le paramètre **pop** est maintenant une **donnée** car ce n'est pas **pop** qui est modifié mais **pop↑.suivant**.

1.2.2.

Première version récursive : il s'agit d'une insertion récursive classique dans une liste triée (cf Tome 2, p. 60).

```
procédure inser (dr pop : pf; d nomfam : ch20);
spécification {pop+ triée} => {(nomfam ∉ pop+, nomfam a été inséré dans
                                pop+) v (nomfam ∈ pop+,pas d'insertion)}
début
    si pop = nil alors
        insertête(pop, nomfam)
    sinon
        si nomfam > pop↑.nom alors
            inser(pop↑.suivant, nomfam)
        sinon
            si nomfam < pop↑.nom alors
                insertête(pop, nomfam);
fin;
```

Deuxième version récursive : il s'agit d'une insertion récursive utilisant le principe de la sentinelle.

```
procédure inser (d pop : pf; d nomfam : ch20);
spécification {pop+ triée} => {(nomfam ∉ pop+, nomfam a été inséré dans
                                pop+) v (nomfam ∈ pop+,pas d'insertion)}
```

```
début
    si pop↑.suivant = nil alors insertête(pop↑.suivant, nomfam)
    sinon  si nomfam > pop↑.suivant↑.nom alors
                inser(pop↑.suivant, nomfam)
            sinon  si nomfam < pop↑.suivant↑.nom alors
                    insertête(pop↑.suivant, nomfam);
fin;
```

Dans le cas d'un schéma récursif, la sentinelle n'apporte rien si ce n'est le passage de paramètre par donnée. On notera également que si l'ajout de la sentinelle respecte l'ordre alphabétique afin que la liste reste triée, la première version peut être utilisée. En conclusion, l'ajout d'une sentinelle en tête n'est intéressant que dans le cas d'un parcours itératif.

1.2.3.

Première version itérative : il s'agit d'une suppression classique dans une liste triée (on s'inspire de l'algorithme de la page 73 du Tome 2). On définit d'abord la procédure auxiliaire **supptête** (cf Tome2, p. 67).

```
procédure supptête (dr pop : pf);
spécification {pop ≠ nil} => {la cellule de tête de pop+ a été supprimée}
var p : pf;
début
    p := pop; pop := pop↑.suivant; laisser(p);
fin;
procédure supp (dr pop : pf; d nomfam : ch20);
spécification {pop+ triée} => {(nomfam ∈ pop+, nomfam a été supprimé
                        de pop+) v (nomfam ∉ pop+,pas de suppression)}
var p : pf; infer : booléen;
début
    si pop ≠ nil alors
        si pop↑.nom = nomfam alors
            {cas particulier : suppression du premier élément}
            supptête(pop)
        sinon
        début
            infer := vrai; p := pop;
            {préservation de pop à cause du passage par donnée résultat}
            tantque (p↑.suivant ≠ nil) et infer faire
                si p↑.suivant↑.nom ≥ nomfam alors
                début
                    infer := faux;
                    si nomfam = p↑.suivant↑.nom alors
                        supptête(p↑.suivant);
                fin
                sinon  p := p↑.suivant;
        fin;
fin;
```

Deuxième version itérative : on utilise une sentinelle en tête de liste afin de supprimer le cas particulier de la suppression en tête.

```
procédure supp (d pop : pf; d nomfam : ch20);
spécification {pop+ triée} => {(nomfam ∈ pop+, nomfam a été supprimé
                        de pop+) ∨ (nomfam ∉ pop+,pas de suppression)}
var infer : booléen;
début
    infer := vrai;
    tantque (pop↑.suivant ≠ nil) et infer faire
        si pop↑.suivant↑.nom ≥ nomfam alors
        début
            infer := faux;
            si nomfam = pop↑.suivant↑.nom alors
                supptête(pop↑.suivant);
        fin
        sinon
            pop := pop↑.suivant;
fin;
```

Comme dans le cas de l'insertion, l'ajout d'une sentinelle en tête permet de simplifier l'algorithme et de passer le paramètre **pop** par **donnée**.

1.2.4. Première version récursive

suppression classique dans une liste triée, on s'inspire de l'algorithme défini page 72 du Tome 2.

```
procédure supp (dr pop : pf; d nomfam : ch20);
spécification {pop+ triée} => {(nomfam ∈ pop+, nomfam a été supprimé
                        de pop+) ∨ (nomfam ∉ pop+,pas de suppression)}
début
    si pop ≠ nil alors
        si pop↑.nom = nomfam alors
            supptête(pop)
        sinon  si pop↑.nom < nomfam alors
                supp(pop↑.suivant, nomfam)
fin;
```

Deuxième version récursive

on utilise une sentinelle en tête.

```
procédure supp (d pop : pf; d nomfam : ch20);
début
    si pop↑.suivant ≠ nil alors
        si pop↑.suivant↑.nom = nomfam alors
            supptête(pop↑.suivant)
        sinon
            si pop↑.suivant↑.nom < nomfam alors
                supp(pop↑.suivant, nomfam)
fin;
```

On remarque, comme dans le cas de l'insertion, que l'ajout d'une sentinelle en tête n'offre pas beaucoup d'intérêt dans le cas d'une suppression récursive. La première version récursive peut être utilisée si la sentinelle respecte l'ordre alphabétique. De nouveau, la sentinelle n'est intéressante que dans le cas d'un parcours sous forme itérative.

2. DEUXIEME PARTIE : LISTE À PLUSIEURS NIVEAUX

2.1. Parcours de listes et comptage d'éléments

2.1.1. Version itérative : parcours classique d'une liste avec comptage d'éléments possédant une certaine propriété (cf Tome 2, p. 25).

```
fonction nbfamillesansparent (d pop : pf) : entier;
spécification { } => {nbfamillesansparent = nombre de familles de pop+
                                          dont les parents sont décédés}
var nb : entier;
début
     nb := 0;
     tantque pop ≠ nil faire
     début
          si pop↑.parent = nil alors nb := nb + 1;
          pop := pop↑.suivant
     fin;
     nbfamillesansparent := nb
fin;
```

Version récursive : (cf Tome 2, p. 26).

```
fonction nbfamillesansparent (d pop : pf) : entier;
spécification { } => {nbfamillesansparent = nombre de familles de pop+
                                          dont les parents sont décédés}
début
     si pop = nil alors
          nbfamillesansparent := 0
     sinon
          si pop↑.parent ≠ nil alors
               nbfamillesansparent := nbfamillesansparent(pop↑.suivant)
          sinon
               nbfamillesansparent:= 1+nbfamillesansparent(pop↑.suivant)
fin;
```

2.1.2.

```
fonction nbcélibat (d pop : pf) : entier;
spécification { } => {(nbcélibat = nombre de familles de pop+ n'ayant plus
                                                   qu'un seul parent}
début
     si pop = nil alors
          nbcélibat := 0
```

```
    sinon
        si pop↑.parent = nil alors
            nbcélibat := nbcélibat(pop↑.suivant)
        sinon
            si pop↑.parent↑.suivant = nil alors
                nbcélibat := 1 + nbcélibat(pop↑.suivant)
            sinon  nbcélibat := nbcélibat(pop↑.suivant)
fin;
```

2.1.3. Version itérative :

Parcours de deux listes dont la seconde dépend de la première. Il faut bien faire l'affectation **enf := pop↑.enfant** et effectuer la parcours avec **enf**. En effet, si ce parcours était réalisé avec **pop**, on obtiendrait l'itération suivante

```
tantque pop↑.enfant ≠ nil faire
début
    nb := nb+1;
    pop↑.enfant := pop↑.enfant↑.suivant ;
fin ;
```

qui aurait pour effet de **détruire** le contenu de **pop↑.enfant** qui vaudrait alors **nil** à la fin de l'itération.

```
fonction nbenfant (d pop : pf) : entier;
spécification { } => {nbenfant  = nombre d'enfants de pop+}
var  enf : pp;  nb : entier;
début
    nb := 0;
    tantque pop ≠ nil faire
    début {parcours liste des familles}
        enf := pop↑.enfant;
        tantque enf ≠ nil faire
        début {parcours liste des enfants}
            nb := nb + 1;
            enf := enf↑.suivant
        fin;
        pop := pop↑.suivant
    fin;
    nbenfant := nb
fin;
```

Version récursive : il faut définir une fonction auxiliaire **nbenffam** correspondant à l'itération interne de l'algorithme précédent.

```
fonction nbenffam (d enf : pp) : entier;
spécification { } => {nbenffam = nombre d'enfants de enf+}
début
    si enf = nil alors nbenffam := 0
    sinon nbenffam := 1 + nbenffam(enf↑.suivant)
fin;
```

```
fonction nbenfant (d pop : pf) : entier;
spécification { } => {nbenfant = nombre d'enfants de pop+}
début
     si pop = nil alors
          nbenfant := 0
     sinon
          nbenfant := nbenffam(pop↑.enfant) + nbenfant(pop↑.suivant)
fin;
```

2.1.4. Version itérative :

```
fonction nbenfnomfam (d pop : pf; d nomfam : ch20) : entier;
spécification {pop+ triée} =>{(nomfam ∈ pop+,nbenfnomfam = nombre
   d'enfants de la famille nomfam) v (nomfam ∉ pop+, nbenfnomfam = 0)}
var fam : pf; nb : entier; enf : pp;
début
     fam := présent(pop, nomfam); nb := 0;
     si fam ≠ nil alors
     début
          enf := fam↑.enfant;
          tantque enf ≠ nil faire
          début {parcours de la liste des enfants nomfam}
               nb := nb + 1;
               enf := enf↑.suivant;
          fin;
     fin;
     nbenfnomfam := nb
fin;
```

Version récursive : on utilise **nbenffam** définie précédemment.

```
fonction nbenfnomfam (d pop : pf; d nomfam : ch20) : entier;
spécification {pop+ triée} =>{(nomfam ∈ pop+,nbenfnomfam = nombre
   d'enfants de la famille nomfam) v (nomfam ∉ pop+, nbenfnomfam = 0)}
var fam : pf;
début
     fam := présent(pop, nomfam);
     si fam = nil alors nbenfnomfam := 0
     sinon nbenfnomfam := nbenffam(fam↑.enfant);
fin;
```

2.1.5. On décompose l'algorithme en deux fonctions récursives : l'une **présentvoit** effectue le parcours des voitures de la famille, l'autre **apparvoit** effectue le parcours des familles en faisant appel à **présentvoit**.

```
fonction présentvoit (d voit : pv; d numéro : ch20) : booléen;
spécification { } => {(voiture numéro ∈ voit+, présentvoit)
                         v (voiture numéro ∉ voit+,¬présentvoit)}
```

```
début
    si voit = nil alors
        présentvoit := faux
    sinon
        si voit↑.numéro = numéro alors
            présentvoit := vrai
        sinon
            présentvoit := présentvoit(voit↑.suivant, numéro);
fin;
```

fonction apparvoit (d pop : pf; d numéro : ch20) : pf;
spécification { } => {(voiture **numéro** ∈ **pop**$^+$,**apparvoit** = adresse de la famille possédant cette voiture)∨ (voiture **numéro** ∉ **pop**$^+$,**apparvoit = nil**)}

```
début
    si pop = nil alors
        apparvoit := nil
    sinon
        si présentvoit(pop↑.voiture, numéro) alors
            apparvoit := pop
        sinon
            apparvoit := apparvoit(pop↑.suivant, numéro);
fin;
```

2.2. Mises à jour d'éléments

2.2.1. On définit d'abord la procédure auxiliaire **instête**.

procédure instête (dr enf : pp; d prénom : ch20; d sexe : car);
spécification { } => {**prénom** de sexe **sexe** a été inséré en tête de **enf**$^+$}

```
var p : pp;
début
    nouveau(p); p↑.prénom := prénom; p↑.sexe := sexe;
    p↑.suivant := enf;
    enf := p;
fin;
```

procédure naissance (d pop : pf; d nomfam, prénom : ch20; d sexe : car);
spécification {**pop**$^+$ **triée**} => {(**nomfam** ∈ **pop**$^+$,**prénom nomfam** du sexe **sexe** a été inséré dans **pop**$^+$) ∨ (**nomfam** ∉ **pop**$^+$,pas d'insertion)}

```
var fam : pf;
début
    fam := présent(pop, nomfam);
    si fam ≠ nil alors
    {nomfam ∈ pop+,insertion du nouvel enfant en tête de fam↑.enfant}
        instête(fam↑.enfant,prénom,sexe)
fin;
```

2.2.2. Définition de la procédure **inser** d'insertion d'un nouvel enfant dans la liste triée des enfants.

```
procédure inser (dr enf : pp; d prénom : ch20; d sexe : car);
spécification {enf+ triée} => {prénom de sexe sexe a été inséré dans enf+ ,
                                                                enf+ triée}
début
    si enf = nil alors
        instête(enf, prénom, sexe)
    sinon
        si enf↑.prénom < prénom alors
            inser(enf↑.suivant, prénom, sexe)
        sinon
            instête(enf, prénom, sexe)
fin;

procédure naissancetrié (d pop : pf; d nomfam, prénom : ch20; d sexe : car);
spécification {pop+ triée , liste des enfants de nomfam ordonnée} =>
              {(nomfam ∈ pop+,prénom  nomfam du sexe sexe a été inséré
                        dans pop+) ∨ (nomfam ∉ pop+,pas d'insertion)}
var fam : pf;
début
    fam := présent(pop, nomfam);
    si fam ≠ nil alors
        inser(fam↑.enfant, prénom, sexe);
fin;
```

2.2.3. Définition de la procédure **supptêtevoit** de suppression de la voiture de tête et de la procédure **suppvoit** de suppression d'une voiture dans la liste des voitures.

```
procédure supptêtevoit(dr voit : pv);
spécification {voit ≠ nil} => {la voiture de tête de voit+ a été supprimée}
var p : pv;
début
    p := voit;
    voit := voit↑.suivant;
    laisser(p);
fin;

fonction suppvoit (d voit : pv ; d  numéro : ch20) : booléen;
spécification { } => {(voiture numéro ∈ voit+,suppvoit ,voiture numéro
                 a été supprimée) ∨ (voiture numéro ∉ voit+,¬suppvoit)}
début
    si voit = nil alors
        suppvoit := faux
    sinon
```

```
        si voit↑.numéro = numéro alors
        début
            supptêtevoit(voit);
            suppvoit := vrai
        fin
        sinon
            suppvoit := suppvoit(voit↑.suivant, numéro)
fin;
```

fonction suppvoiture (d pop : pf; d numéro : ch20) : booléen;
spécification {} => {(voiture **numéro** ∈ **pop+**,**suppvoiture** ,voiture **numéro** **a été supprimée**) ∨ (voiture **numéro** ∉ **pop+**,¬**suppvoiture**)}

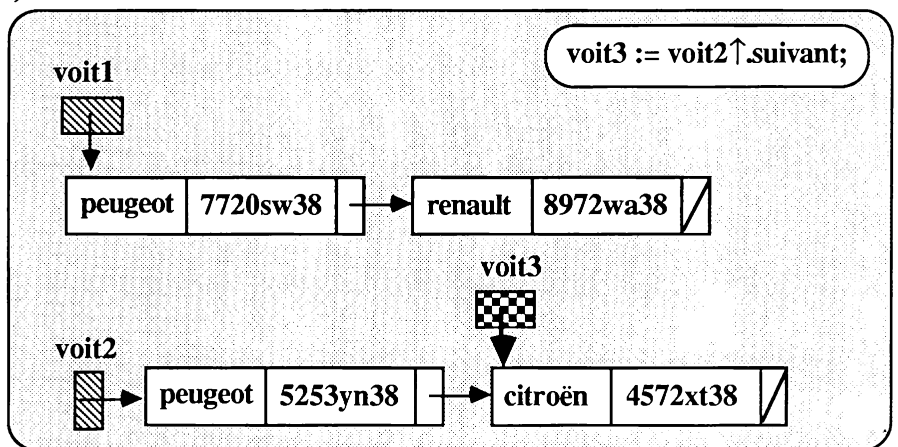
```
début
    si pop = nil alors
        suppvoiture := faux
    sinon
        si suppvoit(pop↑.voiture, numéro) alors
            suppvoiture := vrai
        sinon
            suppvoiture := suppvoiture(pop↑.suivant, numéro)
fin;
```

2.2.4.
Définition de la procédure de changement de propriétaire d'une voiture.

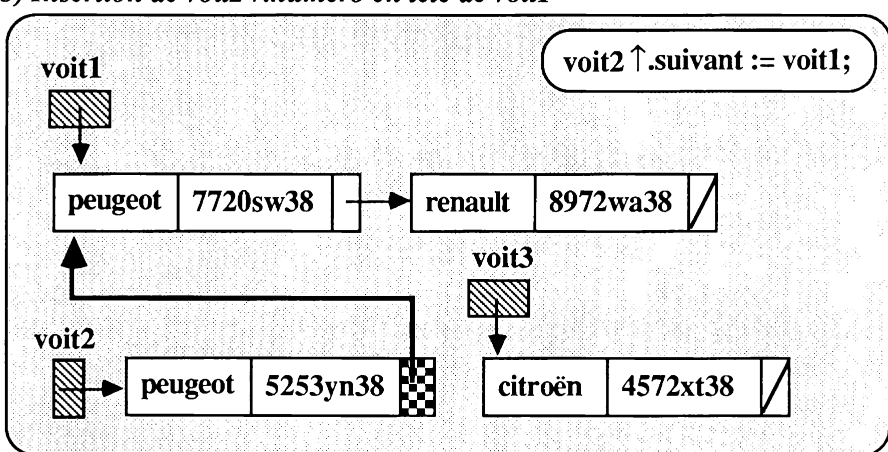
```
procédure changepro (dr voit1, voit2 : pv; d numéro : ch20);
var voit3 : pv;
début
    si voit2 = nil alors
        écrireln('pas de voiture')
    sinon
        si voit2↑.numéro = numéro alors
        début
            {numéro ∈ nom2}
            {protection de l'adresse de la voiture suivant voit2↑.numéro}
            voit3 := voit2↑.suivant;
            voit2↑.suivant := voit1;
            voit1 := voit2;{voit2↑.numéro a été inséré en tête de voit1+}
            voit2 := voit3; {voit2↑.numéro a été supprimé de voit2+}
            {numéro ∈ nom1 , numéro ∉ nom2}
        fin
        sinon
            changepro(voit1, voit2↑.suivant, numéro);
fin;
```

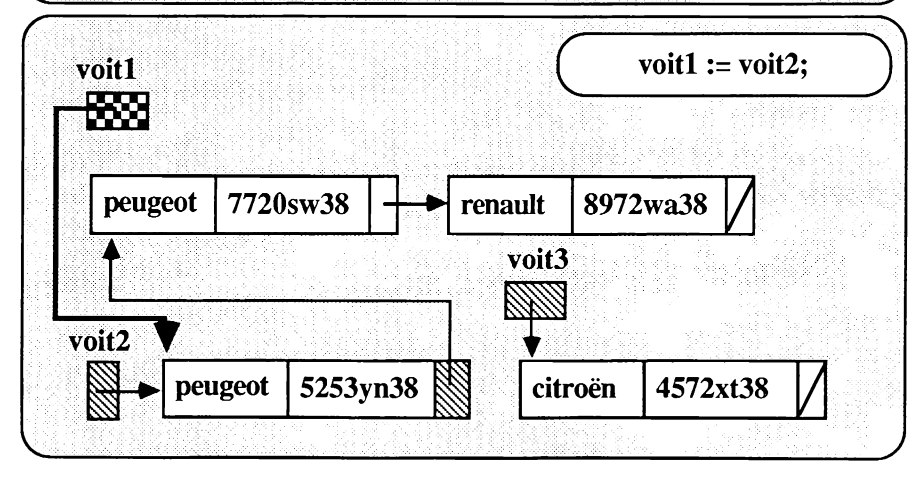
Nous expliquons sur un exemple la procédure **changepro**.
Il s'agit de la voiture **peugeot numéro 5253 yn 38** appartenant à nom2 qui va être supprimée de **voit2+** et insérée en tête de **voit1+**.

a) Protection de l'adresse de la cellule suivant celle d'adresse voit2

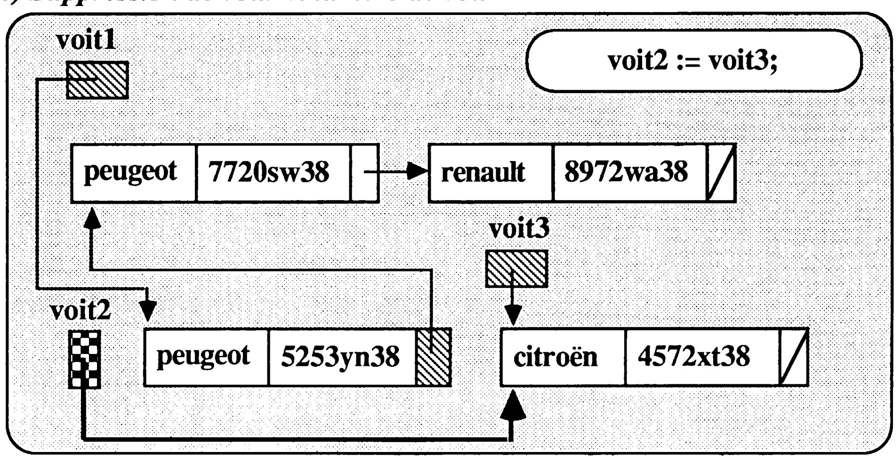


b) Insertion de voit2↑.numéro en tête de voit1+





c) Suppression de voit2↑.numéro de voit2+



procédure achatvoit (d pop : pf ; d nom1, nom2, numéro : ch20);
spécification {**pop+ triée**} => {voiture **numéro** ∈ **famille nom2**,
nom1 ∈ **pop+**, **nom1** a acheté la voiture **numéro** de **nom2**)}
var fam1, fam2 : pf;
début
fam1 := présent(pop, nom1);
si fam1 = nil alors
écrireln('pas d''acheteur')
sinon
début *{nom1 ∈ pop+}*
fam2 := présent(pop, nom2);
si fam2 = nil alors
écrireln('pas de vendeur')
sinon *{nom1 ∈ pop+, nom2 ∈ pop+}*
changepro(fam1↑.voiture, fam2↑.voiture, numéro);
fin;
fin;

4.2. Location d'appartements

Énoncé

L'Office du tourisme d'une petite station de montagne a décidé d'informatiser la gestion des locations saisonnières pour une année. Les structures utilisées seront les suivantes :

1. Une liste chaînée, d'adresse **ot**, des appartements offerts en location. Chaque appartement sera caractérisé dans cette liste par :

- le type de l'appartement (studio, 1 pièce,...), représenté par un code de 2 caractères : ST, T1, T2 ...
- l'indice dans le vecteur **vprop** du nom et de l'adresse du propriétaire
- un pointeur sur une liste de réservations (nil s'il n'y a pas de réservations)

Cette liste est **triée uniquement sur les types d'appartements.**

2- Pour chaque appartement, il existe une liste chaînée des réservations. Les réservations ne pouvant se faire que par semaines entières, les dates de réservations sont remplacées par des numéros de semaine dans l'année. Chaque réservation est caractérisée par :

- le numéro de la semaine du début de la location
- le numéro de la semaine de fin de la location (si la réservation est pour une seule semaine, ces numéros sont identiques)
- le nom du locataire (tronqué à 15 caractères s'il y a lieu)

Ces listes sont triées sur les dates de réservation. Il n'y a pas de "sur-booking" : il ne peut pas y avoir plus d'une réservation pour un appartement pour une semaine donnée. On suppose que les locataires de la station ont tous des noms différents, et qu'aucun n'a effectué plus d'une réservation la même année.

3- Un vecteur **vprop** dans lequel figurent tous les noms et adresses des propriétaires de la station. Dans ce vecteur, chaque propriétaire figure une seule fois, dans un ordre quelconque. Un entier **nbprop** indique le nombre de propriétaires répertoriés. On suppose que tous les propriétaires ont des noms différents.

Déclarations utilisées :

```
type prop      =  structure
                        nom,adresse : ch15
                  fin;
     vectprop  =  tableau [1..50] de prop;
     ptrapp    =  ↑app;
```

```
        ptrres   =  ↑res;
        app      =  structure
                        typeapp : ch2;
                        indprop : entier;
                        reserv : ptrres;
                        appsuiv : ptrapp;
                    fin;
        res      =  structure
                        semdeb, semfin : 1..52;
                        nomloc : ch15;
                        ressuiv : ptrres;
                    fin;
var     vprop    :  vectprop;
        nbprop   :  entier;
        ot       :  ptrapp;
```

Dans l'exemple de la Fig. 1, M. Cohard, demeurant aux Boussardes, possède au moins un studio, actuellement réservé les semaines 2, 3 et 4 pour M. Dupont, et la semaine 6 pour M. Martin.

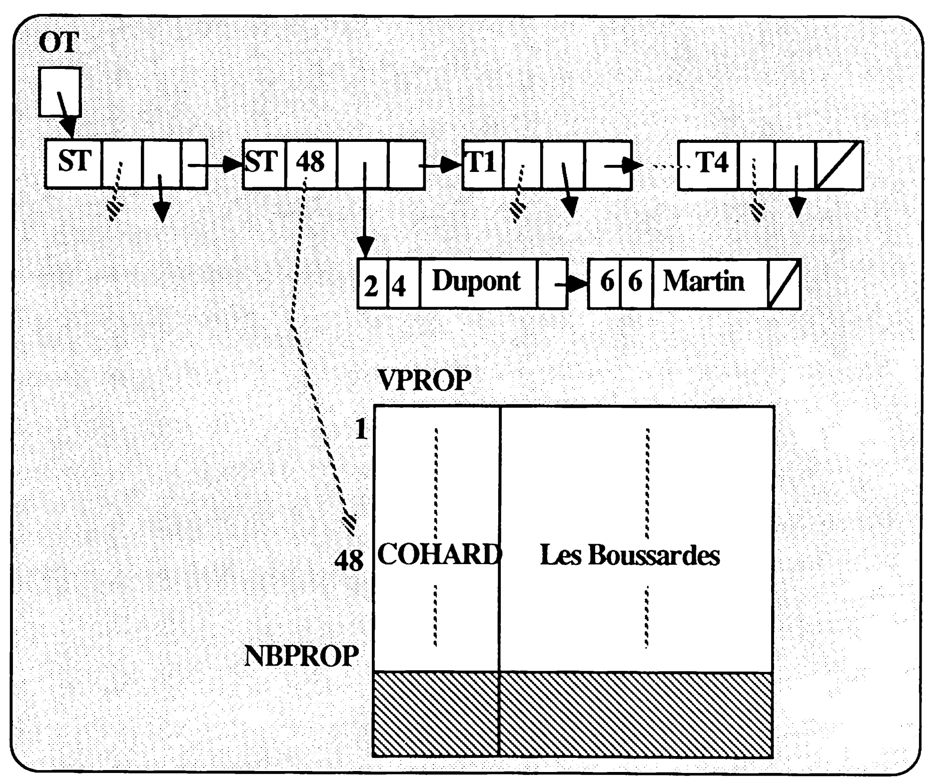


Figure 1

On demande d'écrire les algorithmes suivants, en donnant une version itérative puis une version récursive chaque fois que cela est possible.

On précise que **vprop**, **nbprop** et **ot** sont des variables globales et peuvent donc ne pas figurer dans les en-têtes. Néanmoins, il sera quelquefois utile d'y faire figurer **ot**, afin de pouvoir l'utiliser sans le modifier dans les algorithmes itératifs ; ce sera même nécessaire si l'on veut pouvoir l'utiliser, avec ou sans modifications, dans les algorithmes récursifs.

1. RECHERCHES

1.1.

fonction cherchprop (d nom : ch15) : entier;
spécification { } => {(**nom** $\notin$ **vprop, cherchprop** = 0)
$\vee$ (**vprop [cherchprop].nom** = **nom**)}

1.2.

fonction cherchloc (d ot : ptrapp; d nomloc : ch15) : ptrres;
Cette fonction délivre un pointeur sur la réservation dont le locataire est **nomloc**, ou bien **nil** si aucune réservation n'est trouvée.

2. PARCOURS (COMPTAGES ET LISTES)

2.1.

fonction nbapprop (d ot : ptrapp; r ind : entier) : entier;
Cette fonction délivre le nombre d'appartements qui appartiennent au propriétaire d'indice **ind** dans **vprop**.

2.2.

fonction nbapp (d nom : ch15) : entier;
Cette fonction donne le nombre d'appartements qui appartiennent au propriétaire nommé **nom** , ou bien , par convention, -1 si ce propriétaire est inconnu.

2.3.

fonction nbsemloc (d pr : ptrres) : entier;
Cette fonction délivre le nombre total de semaines de location enregistrées dans la liste de réservations **pr+** .

2.4.

fonction nbvide (d ot : ptrapp) : entier;
Cette fonction délivre le nombre d'appartements qui sont loués moins de 3 semaines par an.

2.5.
procédure listapp (d ot : ptrapp);
Cette procédure imprime la liste complète des appartements, avec pour chacun le type et le nom du propriétaire.

2.6.
procédure listloc (d ot : ptrapp; d numsem : entier);
Cette procédure imprime la liste des locataires pour une semaine de numéro donné **numsem**, avec pour chacun le nom du propriétaire ainsi que le type de l'appartement.

2.7.
fonction libre (d res : ptres ; d numsem : entier) : booléen;
Cette fonction prend la valeur **vrai** si et seulement si l'appartement dont la liste de réservations a pour adresse **res** est libre pendant la semaine de numéro **numsem.**

2.8.
procédure listlibre (d ot : ptrapp; d typeapp : ch2 ; d numsem : entier) ;
Cette procédure imprime les noms et adresses des propriétaires de logements de type **typeapp** qui sont libres pendant la semaine de numéro **numsem.**

3. INSERTIONS

3.1.
fonction insprop (d nom : ch15) : entier ;
Insertion de **nom** dans **vprop** :
- s'il y figure déjà, la fonction délivre son indice dans **vprop**;
- s'il n'y figure pas déjà, la fonction demandera l'adresse, l'insèrera avec le nom dans **vprop**, et délivrera l'indice d'insertion.

3.2.
procédure insapptête (dr pa : ptrapp; d nom : ch15 ; d typeapp : ch2);
Cette procédure insère un nouvel appartement, dont on donne le type **typeapp** et le **nom** du propriétaire, en tête de la liste **pa+**.

3.3.
procédure insapp (dr ot : ptrapp ; d nom : ch15; d typeapp : ch2);
Cette procédure insère un nouvel appartement, dont on donne le type **typeapp** et le **nom** du propriétaire, après les autres appartements du même type, dans la liste **ot+**. Si le nom est nouveau, les mises à jour nécessaires seront effectuées.

3.4.
procédure insertêteres (dr pr : ptrres; d nomloc : ch15;
d sem1,sem2 : entier);
Cette procédure insère une nouvelle réservation en tête de la liste **pr+**. Le nom du locataire,**nomloc**, et les numéros de première et dernière semaine, **sem1** et **sem2**, sont donnés.

3.5.
fonction insère(dr pr : ptrres; d nomloc : ch15;
d sem1,sem2 : entier) : booléen;
Cette fonction tente d'effectuer la réservation, pour le locataire de nom **nomloc**, de la semaine **sem1** à la semaine **sem2**, de l'appartement dont la liste de réservations a pour adresse **pr**. La fonction délivre **vrai** si et seulement si la réservation de cet appartement est possible (période non déjà louée) .

3.6.
fonction réserver (d ot : ptrapp ; d nomloc : ch15; d typapp : ch2;
d sem1,sem2 : entier) : booléen;
Cette fonction tente d'effectuer la réservation au nom de **nomloc** d'un appartement de type **typapp**, de la semaine **sem1** à la semaine **sem2** ; le résultat de la fonction est un booléen qui indique si la réservation a pu être effectuée (il existe un appartement du type souhaité, libre les semaines souhaitées).

4. SUPPRESSIONS

4.1.
fonction suppres (dr pr : ptrres; d nomloc : ch15) : booléen ;
Cette fonction supprime la réservation au nom de **nomloc**, si elle existe, dans la liste **pr+**, et délivre un booléen indiquant si la suppression a été effectuée.

4.2.
procédure annuler (d ot : ptrapp ; d nomloc : ch15);
Cette procédure annule la réservation faite au nom de **nomloc**. S'il n'y avait pas de réservation à ce nom, la procédure imprime un message d'erreur.

5. MISE A JOUR

procédure prolonge (d nomloc : ch15) ;
Cette procédure essaie de prolonger d'une semaine la location au nom de **nomloc**, dans le même appartement. Des messages sont affichés dans tous les cas : prolongation possible (l'appartement est libre pour la semaine souhaitée), prolongation impossible car il n'est pas libre, ou enfin réservation à ce nom inexistante.

Solutions proposées

1. RECHERCHES

1.1. C'est une variante sur la recherche séquentielle dans un vecteur non trié (Tome 1, p. 128). On délivre ici un entier (nul ou non nul) au lieu d'un booléen.

a) Algorithme itératif

```
fonction cherchprop (d nom : ch15) : entier;
spécification { } => {(nom ∉ vprop, cherchprop = 0)
                                    ∨ (vprop [cherchprop].nom = nom)}
var  i : entier;
     trouvé : booléen;
début
     i :=1;
     trouvé := faux;
     cherchprop :=0;
     tantque (i<= nbprop) et non trouvé faire
          si vprop [i].nom = nom alors
               trouvé := vrai
          sinon
               i := i+1;
     si trouvé alors
          cherchprop := i
fin;
```

b) Algorithme récursif :

On est obligé d'ajouter le paramètre "longueur du vecteur" dans l'en-tête.

```
fonction cherchprop (d nbprop : entier ; d nom : ch15) : entier;
spécification { } => {(nom ∉ vprop, cherchprop=0)
                                    ∨ (vprop [cherchprop].nom=nom)}
début
     si nbprop = 0 alors
          cherchprop :=0
     sinon
          si vprop [nbprop].nom = nom alors
               cherchprop :=nbprop
          sinon
               cherchprop :=cherchprop (nbprop-1, nom)
fin;
```

1.2. On utilise la recherche associative dans une liste chaînée non triée (Tome 2, p. 30). Ici il y a deux niveaux de listes, la version récursive oblige donc à écrire deux fonctions séparées.

a) Algorithme itératif

```
fonction cherchloc (d ot : ptrapp;  d nomloc :  ch15)  :  ptrres;
var   trouvé : booléen;
      pr : ptrres;
début
      trouvé :=faux;
      cherchloc :=nil;
      tantque (ot ≠ nil) et non trouvé faire
      début
            pr := ot↑.reserv;
            tantque (pr ≠ nil) et non trouvé faire
                  si pr↑.nomloc = nomloc alors
                        trouvé :=vrai
                  sinon
                        pr := pr↑.ressuiv;
            ot := ot↑.appsuiv
      fin;
      si trouvé alors  cherchloc := pr
fin;
```

b) Algorithme récursif

```
fonction cherchlocres (d pr : ptrres; d nomloc : ch15) : ptrres;
début
      si pr = nil alors
            cherchlocres := nil
      sinon
            si pr↑.nomloc = nomloc alors
                  cherchlocres := pr
            sinon
                  cherchlocres := cherchlocres (pr↑.ressuiv, nomloc)
fin;

fonction cherchloc (d ot : ptrapp;  d nomloc : ch15) : ptrres;
var   pr : ptrres;
début
      si ot = nil alors
            cherchloc := nil
      sinon
      début
            pr := cherchlocres (ot↑.reserv, nomloc);
            si pr ≠ nil alors
                  cherchloc := pr
            sinon
                  cherchloc := cherchloc (ot↑.appsuiv, nomloc)
      fin;
fin;
```

2. PARCOURS (COMPTAGES ET LISTES)

2.1. C'est une application directe de l'algorithme de calcul du nombre d'occurrences d'une valeur dans une liste chaînée (Tome 2, p. 25).

a) Algorithme itératif

```
fonction nbapprop (d ot : ptrapp;  d ind : entier) : entier;
var   n : entier;
début
      n := 0;
      tantque ot ≠ nil faire
      début
            si ot↑.indprop = ind alors
                  n := n+1;
            ot := ot↑.appsuiv
      fin;
      nbapprop := n
fin;
```

b) Algorithme récursif

```
fonction nbapprop (d  ot : ptrapp;  d ind  :  entier) : entier;
début
      si ot = nil alors
            nbapprop := 0
      sinon
            si ot↑.indprop = ind alors
                  nbapprop := 1 + nbapprop (ot↑.appsuiv, ind)
            sinon
                  nbapprop := nbapprop (ot↑.appsuiv, ind)
fin;
```

2.2. On utilise deux fonctions définies ci-dessus : **cherchprop** et **nbapprop**.

```
fonction nbapp (d nom : ch15)  :  entier;
var   ind : entier;
début
      ind := cherchprop (nom);
      si ind = 0 alors
            nbapp := -1
      sinon
            nbapp := nbapprop (ot, ind)
fin;
```

2.3. Il s'agit d'une extension de l'algorithme de calcul de la longueur d'une liste chaînée (Tome 2, p. 21). Au lieu d'ajouter 1 pour chaque élément, on ajoute le nombre de semaines de chaque réservation.

a) Algorithme itératif

```
fonction nbsemloc (d pr : ptrres) : entier;
var   n : entier;
début
      n : = 0;
      tantque pr ≠ nil faire
      début
            n := n + pr↑.semfin - pr↑.semdeb + 1;
            pr := pr↑.ressuiv
      fin;
      nbsemloc := n
fin;
```

b) Algorithme récursif

```
fonction nbsemloc (d pr : ptrres) : entier;
début
      si pr = nil alors  nbsemloc := 0
      sinon
            nbsemloc:=(pr↑.semfin-pr↑.semdeb+1)+ nbsemloc(pr↑.ressuiv)
fin;
```

2.4. C'est de nouveau un calcul de nombre d'occurrences (cf **2.1**). Mais la "valeur" recherchée est calculée au moyen d'un appel de la fonction **nbsemloc**.

a) Algorithme itératif

```
fonction nbvide (d ot : ptrapp) :  entier;
var   nv : entier;
début
      nv := 0;
      tantque ot ≠ nil faire
      début
            si nbsemloc (ot↑.reserv) < 3 alors  nv := nv + 1;
            ot := ot↑.appsuiv
      fin;
      nbvide := nv
fin;
```

b) Algorithme récursif

```
fonction nbvide (ot : ptrapp) : entier;
début
      si ot = nil alors  nbvide := 0
      sinon
            si nbsemloc (ot↑.reserv) < 3 alors
                  nbvide := 1 + nbvide (ot↑.appsuiv)
            sinon nbvide := nbvide (ot↑.appsuiv)
fin;
```

2.5. C'est le parcours simple d'une liste chaînée, avec écriture de chaque élément.
a) Algorithme itératif

```
procédure listapp (d ot : ptrapp);
début
    tantque ot ≠ nil faire
    début
        écrireln (ot↑.typeapp,  vprop[ot↑.indprop].nom);
        ot := ot↑.appsuiv
    fin
fin;
```

b) Algorithme récursif

```
procédure listapp (d ot : ptrapp);
début
    si ot ≠ nil alors
    début
        écrireln (ot↑.typeapp,  vprop [ot↑.indprop].nom);
        listapp (ot↑.appsuiv)
    fin
fin;
```

2.6. L'algorithme est un parcours de la structure à deux niveaux de listes chaînées.
Algorithme itératif :

```
procédure listloc (d ot : ptrapp;  d numsem : entier);
var  pr : ptrres;
    trouvé : booléen;
début
    tantque ot ≠ nil faire
    début
        pr := ot↑.reserv;
        trouvé := faux;
        tantque (pr ≠ nil) et non trouvé faire
            si pr↑.semfin < numsem alors pr := pr↑.ressuiv
            sinon
            début {numsem ≤ pr↑.semfin}
                trouvé := vrai;
                si pr↑.semdeb ≤ numsem alors
                    {pr↑.semdeb ≤ numsem ≤ pr↑.semfin}
                    écrireln (pr↑.nomloc,
                                    vprop [ot↑.indprop].nom, ot↑.typeapp)
            fin;
        ot := ot↑.appsuiv
    fin
fin;
```

La version récursive serait possible, mais entraînerait des passages de paramètres assez lourds pour mémoriser l'indice du propriétaire et le type de l'appartement.

2.7. Il s'agit d'une variante, un peu plus complexe, de la recherche d'une valeur dans une liste triée (Tome 2, p. 34).

a) Algorithme itératif

```
fonction libre (d res : ptrres;  d numsem : entier) : booléen;
var   fini : booléen;
début
      libre := vrai;
      fini := faux;
      tantque (res ≠ nil) et non fini faire
            si res↑.semdeb > numsem alors
                  fini := vrai
            sinon
                  si res↑.semfin ≥ numsem alors
                  début {res↑.semdeb ≤ numsem ≤ res↑.semfin}
                        libre := faux;
                        fini := vrai
                  fin
                  sinon
                        res := res↑.ressuiv
fin;
```

b) Algorithme récursif

```
fonction libre (d res :  ptrres; d numsem : entier) : booléen;
début
      si res = nil alors  libre := vrai
      sinon
            si res↑.semdeb > numsem alors  libre  := vrai
            sinon
                  si res↑.semfin ≥ numsem alors libre := faux
                  sinon libre := libre(res↑.ressuiv, numsem)
fin;
```

2.8. Ici encore, on recherche toutes les occurrences d'une valeur dans une liste triée.

a) Algorithme itératif

```
procédure listlibre (d ot : ptrapp;  d typeapp :  ch2;  d numsem : entier);
var   fini : booléen;
      indprop : entier;
début
```

```
    fini := faux;
    tantque (ot ≠ nil) et non fini faire
        si ot↑.typeapp < typeapp alors  ot := ot↑.appsuiv
        sinon
            si ot↑.typeapp = typeapp alors
            début
                si libre (ot↑.reserv, numsem) alors
                    écrireln (vprop [ot↑.indprop].nom,
                                        vprop [ot↑.indprop].adresse
                ot := ot↑.appsuiv
            fin
            sinon fini := vrai
fin;
```

b) Algorithme récursif

```
procédure listlibre (d ot : ptrapp; d typeapp : ch2; d numsem : entier);
début
    si ot ≠ nil alors
        si ot↑.typeapp < typeapp alors
            listlibre  (ot↑.appsuiv, typeapp, numsem)
        sinon
            si ot↑.typeapp = typeapp alors
            début
                si libre (ot↑.reserv, numsem) alors
                        écrireln (vprop [ot↑.indprop].nom,
                                        vprop [ot↑.indprop].adresse
                listlibre (ot↑.appsuiv, typeapp, numsem)
            fin;
fin;
```

3. INSERTIONS

3.1. Le vecteur **vprop** n'est pas trié, on se contente donc d'insérer le coordonnées des nouveaux propriétaires à la fin du vecteur.

```
fonction insprop (d nom : ch15) : entier;
var  i : entier;  adresse : ch15;
début
    i := cherchprop (nom);
    si i = 0 alors
    début
        nbprop := nbprop+1;
        vprop [nbprop].nom := nom;
        écrire ('adresse de ',nom,' ?');  lireln (vprop [nbprop].adresse);
        insprop := nbprop
    fin
    sinon  insprop := i
fin;
```

3.2. C'est l'insertion d'une nouvelle cellule en tête d'une liste chaînée ; il faut penser à initialiser tous les champs de la nouvelle cellule, soit à l'aide des données, soit avec une valeur "neutre" telle que **nil** pour un pointeur.

```
procédure insapptête (dr pa : ptrapp;  d nom : ch15;  d typeapp : ch2);
var   p : ptrapp;
début
      nouveau (p);
      p↑.indprop := insprop (nom);
      p↑.typeapp := typeapp;
      p↑.appsuiv := pa; p↑.reserv := nil;
      pa := p
fin;
```

3.3. Insertion d'un élément dans une liste triée (Tome 2, p. 58)

a) Algorithme itératif

```
procédure insapp (dr ot : ptrapp;  d nom : ch15;  d typeapp : ch2);
var   app, préced : ptrapp;
      super : booléen;
début
      si ot = nil alors  insapptête (ot, nom, typeapp)
      sinon
            si typeapp < ot↑.typeapp alors  insapptête (ot, nom, typeapp)
            sinon
            début
                  préced := ot;
                  app := ot↑.appsuiv;
                  super := vrai;
                  tantque (app ≠ nil) et super faire
                        si typeapp ≥  app↑.typeapp alors
                        début
                              préced := app;
                              app := app↑.appsuiv
                        fin
                        sinon  {typeapp <  app↑.typeapp}   super := faux;
                  insapptête (préced↑.appsuiv, nom, typeapp)
            fin {insertion à la fin des appartements de même type}
fin;
```

b) Algorithme récursif (préférable en raison de sa simplicité)

```
procédure insapp (dr ot : ptrapp;  d nom : ch15;  d typeapp : ch2);
début
      si ot = nil alors insapptête (ot, nom, typeapp)
      sinon
            si ot↑. typeapp > typeapp alors  insapptête (ot, nom,typeapp)
            sinon  insapp (ot↑.appsuiv, nom, typeapp)
fin;
```

3.4.

```
procédure insertêteres (dr pr : ptrres;  d nomloc : ch15;
                                        d sem1, sem2 : entier);
var q : ptrres;
début
     nouveau (q);
     q↑.ressuiv := pr;
     q↑.nomloc := nomloc;
     q↑.semdeb := sem1;
     q↑.semfin := sem2;
     pr :=q
fin;
```

3.5. Il s'agit encore d'une variante de l'insertion d'un élément dans une liste triée. Nous ne donnons que la version récursive, car la version itérative serait particulièrement lourde.

```
fonction insère (dr pr : ptrres;  d nomloc : ch15;  d sem1, sem2 : entier)
                                                          : booléen;
début
     si pr = nil alors
     début
          insertêteres (pr, nomloc, sem1, sem2);
          insère := vrai
      fin
     sinon
          si pr↑.semdeb > sem2 alors
          début
               insertêteres (pr, nomloc, sem1, sem2);
               insère := vrai
          fin
          sinon
               si pr↑.semfin < sem1 alors
                    insère := insère (pr↑.ressuiv, nomloc, sem1, sem2)
               sinon
                    insère := faux
fin;
```

3.6. On appelle ici la fonction **insère** définie ci-dessus, à l'intérieur d'un algorithme qui est semblable à la recherche d'une valeur dans une liste triée.

a) Algorithme itératif

```
fonction réserver(d ot : ptrapp;  d nomloc : ch15; d typeapp : ch2;
                                   d sem1, sem2 : entier) : booléen;
var  pr : ptrres;
     fintype, trouvé : booléen;
```

```
début
    fintype := faux;
    trouvé := faux;
    tantque (ot ≠ nil) et non trouvé et non fintype faire
        si ot↑.typeapp > typeapp alors
            fintype := vrai
        sinon
            si ot↑.typeapp < typeapp alors
                ot := ot↑. appsuiv
            sinon
                si insère (ot↑.reserv, nomloc, sem1, sem2) alors
                    trouvé := vrai
                sinon
                    ot := ot↑.appsuiv;
    réserver := trouvé
fin;
```

b) Algorithme récursif

```
fonction réserver (d ot : ptrapp;  d nomloc : ch15;  d typeapp : ch2;
                                        d  sem1, sem2 : entier) : booléen;
début
    si ot=nil alors
        réserver:= faux
    sinon
        si ot↑.typeapp > typeapp alors
            réserver:= faux
        sinon
            si ot↑.typeapp < typeapp alors
                réserver :=
                réserver (ot↑.appsuiv, nomloc, typeapp, sem1, sem2)
            sinon
                si insère (ot↑reserv, nomloc, sem1, sem2) alors
                    réserver := vrai
                sinon  réserver :=
                réserver (ot↑.appsuiv, nomloc, typeapp, sem1, sem2)
fin;
```

4. SUPPRESSIONS

4.1. Il s'agit ici de la suppression d'une valeur dans une liste non triée (Tome 2, p. 70). Nous ne donnons que la version récursive en raison de sa simplicité.

```
fonction suppres (dr pr : ptrres;  d nomloc : ch15) : booléen;
var   qr : ptrres;
```

```
début
    si pr = nil alors suppres := faux
    sinon
        si pr↑.nomloc = nomloc alors
        début
            qr := pr;
            pr := pr↑.ressuiv;
            laisser (qr);
            suppres := vrai
        fin
        sinon
            suppres := suppres(pr↑.ressuiv, nomloc)
fin;
```

4.2. Appel de la fonction précédente, dans un parcours de liste non triée.

a) Algorithme itératif

```
procédure annuler (d ot : ptrapp; d nomloc : ch15);
var  fini : booléen;
début
    fini := faux;
    tantque (ot ≠ nil) et non fini faire
        si suppres (ot↑.reserv, nomloc) alors
            fini := vrai
        sinon
            ot := ot↑.appsuiv;
    si ot = nil alors
        écrireln ('erreur sur le nom du locataire: ', nomloc)
fin;
```

b) Algorithme récursif

```
procédure annuler (d ot : ptrapp; d nomloc : ch15);
début
    si ot=nil alors
        écrireln ('erreur sur le nom du locataire :  ', nomloc)
    sinon
        si non suppres (ot↑.reserv, nomloc) alors
            annuler (ot↑.appsuiv, nomloc)
fin;
```

5. MISE A JOUR

Dans cet algorithme, on commence par appeler la fonction **cherchloc** pour obtenir un pointeur sur la réservation initiale, si elle existe. Puis on regarde si la semaine qui suit la dernière semaine enregistrée est libre (pas de réservation ultérieure ou bien réservation ultérieure commençant plus tard).

```
procédure prolonge (d nomloc : ch15);
var   pr : ptrres;
      sem : entier;
début
      pr := cherchloc (ot, nomloc);
      si pr ≠ nil alors
      début
            sem := pr↑.semfin + 1;
            si pr↑.ressuiv = nil alors
            début
                  pr↑.semfin := sem;
                  écrireln ('prolongation enregistrée')
            fin
            sinon
                  si pr↑.ressuiv↑.semdeb > sem alors
                  début
                        pr↑.semfin := sem;
                        écrireln ('prolongation enregistrée')
                  fin
                  sinon
                        écrireln ('prolongation impossible')
      fin
      sinon
            écrireln ('erreur sur le nom du locataire : ', nomloc)
fin;
```

4.3. Location de skis

Énoncé

On dispose de la structure suivante (voir la figure) :
mat est un pointeur sur une liste dont chaque cellule représente une paire de skis.

Chaque cellule de cette liste est constituée de trois champs :
. le premier contient la **taille** de la paire de skis,
. le deuxième **loc** est un pointeur sur une sous-liste de réservations fournissant des informations sur la location de cette paire de skis,
. le troisième **skisuiv** est un pointeur sur la paire de skis suivante.

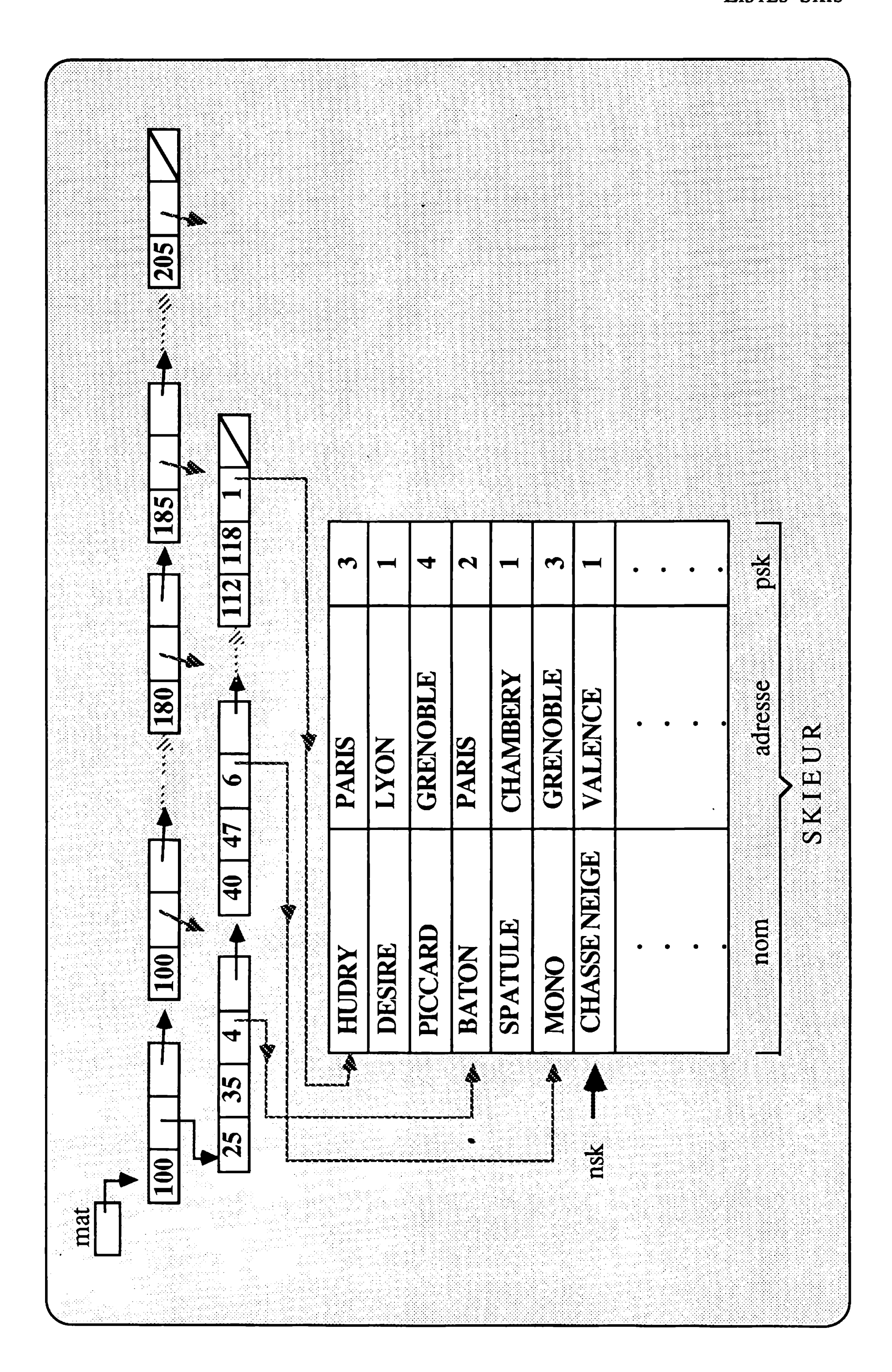
Cette liste est **ordonnée par ordre croissant sur la taille** des skis.

Chaque cellule des sous-listes de réservations est constituée de quatre champs :
. les deux premiers **deb** et **fin** indiquent respectivement le 1er et le dernier jour de location de la paire de skis,
. le troisième **ind** est un indice repérant un élément du vecteur skieur,
. le quatrième **ressuiv** est un pointeur sur la cellule suivante.
Chaque sous-liste de réservations **est ordonnée sur les jours de location.**

Le vecteur **skieur[1..nsk]** (nsk > 0) est composé d'éléments possédant trois champs :
. les deux premiers **nom** et **adresse** indiquent le nom et l'adresse du skieur ayant loué des paires de skis,
. le troisième **psk** précise le nombre de paires de skis louées.

N.B. une même paire de skis louée par la même personne à n périodes différentes compte pour n; ce nombre n'est jamais nul.

Exemple :
Pour la première paire de skis de taille 100,
. BATON de PARIS (qui a loué 2 paires) l'a louée du 25ème au 35ème jour (11 jours de location),
. MONO de GRENOBLE (qui a loué 3 paires) l'a louée du 40ème au 47ème jour,
. HUDRY de PARIS (qui a loué 3 paires) l'a louée du 112ème au 118ème jour.
. . . .


mat
100
100
180
185
205
25
35
4
40
47
6
112
118
1
HUDRY
PARIS
3
DESIRE
LYON
1
PICCARD
GRENOBLE
4
BATON
PARIS
2
SPATULE
CHAMBERY
1
MONO
GRENOBLE
3
CHASSE NEIGE
VALENCE
1
nsk
nom
adresse
psk
SKIEUR

On utilisera les déclarations suivantes :

```
type
        ptski   =  ↑pairski ;
        pairski =  structure
                        taille : entier ;
                        loc : ptreserv ;
                        skisuiv : ptski
                   fin ;
        ptreserv = ↑reserv ;
        reserv  =  structure
                        deb : entier ;
                        fin : entier ;
                        ind : entier ;
                        ressuiv : ptreserv
                   fin ;
        élément =  structure
                        nom : chaîne20 ;
                        adresse : chaîne30 ;
                        psk : entier
                   fin ;
        vecteur =  tableau [1..1000] de élément ;

        elem    =  structure
                        de, fi, t : entier
                   fin ;
        vectelem = tableau [1..100] de elem ;

var mat : ptski ; skieur : vecteur ; nsk : entier ; res : vectelem ...
```

La saison commence le 1er décembre (jour n° 1) et se termine le 30 avril (jour n° 151 : année non bissextile).

1. PARCOURS DE LISTES

1.1. On veut connaître le nombre de paires de skis présentes dans la structure **mat↑**. Écrire, **sous forme itérative**, une
fonction nbskis (d mat : ptski) : entier ;
spécification { } => {**nbskis** = nombre total de paires de skis de **mat↑**}

1.2. On désire connaître l'adresse de la première cellule contenant une paire de skis de taille **t**. Écrire, **sous forme récursive**, une
fonction skit (d mat : ptski ; d t : entier) : ptski ;
spécification {**mat↑ trié**} => {(**skit** = adresse de la première cellule contenant une paire de skis de taille **t**) ∨ (**skit** = **nil** si cette cellule n'existe pas)}

1.3. Écrire, **sous forme itérative**, une
fonction nbjloc (d ski : ptski) : entier ;
spécification {**ski ≠ nil**} => {**nbjloc** = nombre de jours de location de **ski↑.loc+**}
qui délivre le nombre de jours de location d'une paire de skis.

1.4. Écrire la fonction précédente **sous forme récursive**.

2. PARCOURS DE VECTEURS

On veut connaître l'indice de **nom** dans le vecteur skieur[1..nsk]. Écrire, **sous forme récursive,** une
fonction cherchskieur (d skieur : vecteur ; d nsk : entier ; d nom : chaîne20) :entier ;
spécification { } => {(**cherchskieur** = indice de **nom** dans **skieur [1..nsk]**)
v (**cherchskieur = 0 , nom** ∉ **skieur [1..nsk]**)}
On notera que tous les skieurs ont des noms différents.

3. INSERTIONS D'ÉLÉMENTS

3.1. On définit tout d'abord une procédure auxiliaire **insertête** que l'on utilisera dans la question suivante. Écrire la
procédure insertête (dr locski : ptreserv ; d deb, fin, ind : entier) ;
spécification { } ==> {la cellule contenant **deb**, **fin** et **ind** a été insérée en tête de **locski+**}

3.2. On souhaite louer une paire de skis déterminée, telle que **premloc** soit un pointeur sur sa liste de réservations. On connaît les jours de début **deb**, de fin **fin** et l'indice **ind** du skieur.
Écrire, **sous forme récursive**, la
fonction location (dr premloc : ptreserv ; d deb, fin, ind : entier) : booléen ;
spécification {**premloc+ trié**} => {(**location**, la location a été réalisée)
v (**non location**, la location n'a pas été réalisée)}
On utilisera **insertête** et on supposera que : $1 \leq deb \leq fin \leq 151$.

3.3. Écrire une
fonction insertskieur (dr skieur : vecteur; dr nsk : entier; d nom : chaîne20 ; d adresse : chaîne30) : entier ;
spécification { } => {**nom** ∈ **skieur [1..nsk]**,
insertskieur = indice de **nom** dans **skieur [1..nsk]**}
On suppose que tous les skieurs ont un nom différent.

3.4. Monsieur **nom** d'adresse **adresse** désire louer une paire de skis de taille **t** du jour **de** au jour **fi** (fi ≥ de). Écrire une

procédure réservation (d mat : ptski; d nom : chaîne20 ; dr skieur : vecteur ; dr nsk : entier; d adresse : chaîne30; d de, fi, t : entier; r possible : booléen) ;
spécification {**mat**+ **trié**} => {(**possible**, la réservation a été effectuée)
v (**non possible**, la réservation n'a pas été effectuée)}

3.5. Monsieur **nom** d'adresse **adresse** désire louer des paires de skis (pour toute la famille et plusieurs séjours). On dispose d'un vecteur **res[1..nr]** de type vectelem dont chaque élément est composé de trois champs : **de, fi, t.** Écrire une
procédure réservations (d mat : ptski; d nom : chaîne20; dr skieur : vecteur; dr nsk : entier ; d adresse : chaîne30 ; d res : vectelem ; d nr : entier) ;
spécification {**mat**+ **trié**} => {les réservations demandées ont été réalisées si possible et on a fait imprimer des messages adéquats}

4. SUPPRESSIONS D'ÉLÉMENTS

Monsieur **nom** a un empêchement et il annule toutes les locations qu'il a effectuées. Écrire une procédure d'annulation.

Solutions proposées

1. PARCOURS DE LISTES

1.1. Parcours itératif de listes (cf Tome 2, p. 22).

```
fonction nbskis (d mat : ptski) : entier ;
spécification { } => {nbskis = nombre total de paires de skis de mat+}
var nb : entier ;
début
    nb := 0 ;
    tantque mat ≠ nil faire
    début
        nb := nb + 1 ;
        mat := mat↑.skisuiv
    fin;
    nbskis := nb
fin;
```

1.2. Accès associatif, sous forme récursive, dans une liste triée (cf Tome 2, p. 34).

fonction skit (d mat : ptski ; d t : entier) : ptski ;
spécification {**mat**+ **trié**} => {(**skit** = adresse de la première cellule
contenant une paire de skis de taille **t**)
v (**skit** = **nil** si cette cellule n'existe pas)}

```
début
      si mat = nil alors
            skit := nil
      sinon
            si mat^.taille = t alors
                  skit := mat
            sinon
                  si mat^.taille < t alors
                        skit := skit(mat^.skisuiv,t)
                  sinon
                        skit := nil
fin ;
```

1.3. Parcours itératif de listes avec comptage.

```
fonction nbjloc (d ski : ptski) : entier ;
spécification {ski ≠ nil} => {nbjloc = nombre de jours de location de
                                                            ski↑.loc+}
var loc : ptreserv; nb : entier;
début
      loc := ski^.loc;
      nb := 0;
      tantque loc ≠ nil faire
      début
            nb := nb + loc^.fin - loc^.deb + 1;
            loc := loc^.ressuiv
      fin;
      nbjloc := nb
fin;
```

1.4. On définit une fonction auxiliaire **jloc** équivalente à l'itération précédente. Ensuite, la fonction **nbjloc1** fait appel à la fonction **jloc** avec le paramètre **ski^.loc.**

```
fonction jloc (d locat : ptreserv) : entier;
spécification { } => {jloc = nombre de jours de location de locat+}
début
      si locat = nil alors
            jloc := 0
      sinon jloc := locat^.fin - locat^.deb + 1 + jloc(locat^.ressuiv)
fin;

fonction nbjloc1 (d ski : ptski) : entier ;
spécification {ski ≠ nil} => {nbjloc1 = nombre de jours de location
                                                            ski^.loc+}
début
      nbjloc1 := jloc(ski^.loc)
fin;
```

2. PARCOURS DE VECTEURS

Parcours récursif du vecteur à partir de la fin (cf Tome 1, p. 132).

```
fonction cherchskieur (d skieur : vecteur ; d nsk : entier ;
                                        d nom : chaîne20) :entier ;
spécification { } => {(cherchskieur = indice de nom dans skieur [1..nsk])
                         ∨ (cherchskieur = 0 , nom ∉ skieur [1..nsk]}
début
    si nsk = 0 alors cherchskieur := 0
    sinon si skieur[nsk].nom = nom alors cherchskieur := nsk
          sinon cherchskieur := cherchskieur(skieur, nsk-1, nom)
fin;
```

3. INSERTIONS D'ÉLÉMENTS

3.1. On s'inspire de **insertête** définie dans le cours (cf Tome 2, p. 46).

```
procédure insertête (dr locski : ptreserv ; d deb, fin, ind : entier) ;
spécification { } ==> {la cellule contenant deb, fin et ind a été
                                        insérée en tête de locski+}
var p : ptreserv;
début
    nouveau(p); p↑.deb := deb; p↑.fin := fin; p↑.ind := ind;
    p↑.ressuiv := lockski; lockski := p
fin;
```

3.2. On s'inspire de **insertri** définie dans le cours (cf Tome 2, p. 60).

```
fonction location (dr premloc : ptreserv ; d deb, fin, ind : entier) : booléen ;
spécification {premloc+ trié} => {(location, la location a été réalisée)
                       ∨ (non location, la location n'a pas été réalisée)}
début
    si premloc = nil alors
    début
        insertête (premloc,deb,fin,ind);
        location := vrai
    fin
    sinon
        si premloc↑.deb > fin alors
        début
            insertête (premloc,deb,fin,ind);
            location := vrai
        fin
        sinon
            si premloc↑.fin < deb alors
                location := location (premloc↑.ressuiv,deb,fin,ind)
            sinon
                location := faux
fin;
```

3.3. Après avoir cherché l'indice de **nom** dans **skieur**, on insère, si nécessaire, **nom** à la fin du vecteur en mettant à jour les champs de skieur.

```
fonction insertskieur (dr skieur : vecteur; dr nsk : entier; d nom : chaîne20 ;
                                            d adresse : chaîne30) : entier ;
spécification { } =>       {nom ∈ skieur [1..nsk],
                            insertskieur = indice de nom dans skieur [1..nsk]}
var m : entier;
début {recherche de l'indice de nom dans skieur [1..nsk]}
    m := cherchskieur(skieur, nsk, nom);
    si m ≠ 0 alors {nom ∈ skieur [1..nsk]} insertskieur := m
    sinon
    début {nom ∉ skieur [1..nsk],insertion de nom dans skieur [1..nsk]}
        nsk := nsk + 1 ; skieur[nsk].nom := nom ;
        skieur[nsk].adresse := adresse ; skieur[nsk].psk := 0 ;
        insertskieur := nsk
    fin
fin;
```

3.4.

```
procédure réservation (d mat : ptski; d nom : chaîne20 ; dr skieur : vecteur ;
dr nsk : entier; d adresse : chaîne30; d de, fi, t : entier; r possible : booléen) ;
spécification {mat+ trié} => {(possible, la réservation a été effectuée)
                        v (non possible, la réservation n'a pas été effectuée)}
var égal,trouvé:booléen; pts:ptski; ind:entier;
début
    pts := skit(mat,t); {pts = adresse de la première cellule de taille t}
    trouvé := faux;
    égal := vrai;
    ind := insertskieur(skieur,nsk,nom, adresse);
    {nom a été inséré si nécessaire , insertskieur = indice de nom}
    tantque(pts ≠ nil) et (non trouvé) et égal faire
        si pts↑.taille > t alors égal := faux
        sinon
            si location(pts↑.loc, de, fi, ind) alors
            début {la location a été réalisée}
                trouvé := vrai;
                skieur[ind].psk := skieur[ind].psk + 1 ;
            fin
            sinon {la location n'a pas été réalisée}
                pts := pts↑.skisuiv;
    si (non trouvé) et (skieur[nsk].psk = 0) alors
        {la location n'a pas été réalisée , le nom du skieur éventuellement
                                inséré à la fin du vecteur est supprimé}
        nsk := nsk-1;
    possible := trouvé
fin;
```

3.5. Appel de la procédure précédente pour tous les éléments du vecteur **res**.

```
procédure réservations (d mat : ptski; d nom : chaîne20; dr skieur : vecteur;
        dr nsk : entier ; d adresse : chaîne30 ; d res : vectelem ; d nr : entier) ;
spécification {mat+ trié} => {les réservations demandées ont été réalisées si
                    possible et on a fait imprimer des messages adéquats}
var possible:booléen; i:entier;
début
    pour i := 1 haut nr faire
    début
        réservation (mat,nom,skieur,nsk,adresse,res[i].de,res[i].fi,res[i].t
                                                            ,possible);
        si possible alors
            écrire ('réservation du',res[i].de:4,' au', res[i].fi:4,
                    ' de la paire de ski de taille', res[i].t:4, ' effectuée')
        sinon
            écrire ('désolé, réservation du',res[i].de:4,' au', res[i].fi:4,
                    ' de la paire de ski de taille', res[i].t:4, ' impossible')
    fin
fin;
```

4. SUPPRESSIONS D'ÉLÉMENTS

On définit d'abord une procédure auxiliaire, sous forme récursive, de suppression de toutes les paires de skis louées par le skieur **skieur[ind].nom** dans la liste de réservation **p+**.

```
procédure supp (dr p: ptreserv;  d ind:entier;  dr psk: entier);
spécification  { }  ==>  {toutes les paires de skis louées par skieur[ind].nom
                                                  ont été supprimées}
var q: ptreserv;
début
    si (psk > 0) et (p ≠ nil) alors
    début
        si p↑.ind = ind alors
        début {suppression de la réservation}
            q := p;
            p := p↑.ressuiv;
            laisser(q);
            psk := psk-1;
            supp(p,ind,psk)
        fin
        sinon
            supp(p↑.ressuiv, ind, psk)
    fin
fin;
```

```
procédure annuler (d mat : ptski; d nom : chaîne20; dr skieur : vecteur;
                                    dr nsk : entier; r possible : booléen);
var p : ptreserv; psk , ind : entier; q : ptski;
début
    ind := cherchskieur(skieur,nsk,nom); {ind = indice de nom}
    si ind = 0 alors {nom n'est pas présent}
        possible := faux
    sinon {nom est présent}
    début
        psk := skieur[ind].psk; {psk = nombre de paires de skis louées}
        q:=mat;
        tantque (mat ≠ nil) et (psk > 0) faire
        début {suppression des paires de skis louées par nom}
            supp(mat↑.loc,ind,psk);
            mat := mat^.skisuiv
        fin;
        si ind ≠ nsk alors
        début
          {suppression de nom par copie dans skieur[ind] de skieur[nsk]}
            skieur[ind] := skieur[nsk];
            mat := q;
            tantque mat ≠ nil faire
            début
                {mise à jour de l'indice dans la liste des réservations}
                p := mat↑.loc;
                tantque p ≠ nil faire
                début
                    si p↑.ind = nsk alors
                        p↑.ind := ind;
                    p := p↑.ressuiv
                fin;
                mat := mat↑.skisuiv
            fin
        fin;
        nsk := nsk-1; {suppression du dernier par mise à jour de nsk}
        possible := vrai
    fin
fin;
```

5

STRUCTURES LINÉAIRES PARTICULIERES

Éditeur de texte

Énoncé

On souhaite réaliser un éditeur de texte rudimentaire, qui opère exclusivement par lignes entières et qui offre au moins les fonctionnalités suivantes :
- initialisation de l'éditeur avec un texte vide,
- insertion d'une nouvelle ligne à la fin du texte (vide ou non),
- insertion d'une nouvelle ligne avant ou après une ligne donnée,
- destruction d'une ligne,
- déplacement, copie, effacement d'un bloc (ensemble consécutif de lignes),
- chargement d'un fichier dans l'éditeur,
- rangement du contenu de l'éditeur dans un fichier.

L'interface utilisateur de l'éditeur, c'est-à-dire l'interpréteur de commandes, ne sera pas traité dans cet exercice. On se bornera à proposer des structures de données pour l'éditeur, et à étudier les algorithmes qui permettent de réaliser les fonctionnalités énumérées ci-dessus, dans le contexte de ces structures de données.

Chaque ligne sera représentée dans une chaîne de 80 caractères, et les fichiers correspondants seront des fichiers de type *text* . L'éditeur sera une liste linéaire doublement chaînée de cellules, composées chacune d'une ligne de texte et de deux pointeurs, l'un vers la ligne précédente et l'autre vers la ligne suivante.

Nous traiterons deux versions de la structure de texte, qui utiliseront les mêmes déclarations de type :

```
type  ch80      =  chaîne80;
      pointeur  =  ↑lignetext;
      lignetext =  structure
                        prec : pointeur;
                        ligne : ch80;
                        suiv : pointeur
                   fin;
```

1. PREMIERE VERSION DE L'ÉDITEUR : accès au texte par des pointeurs de début et de fin (fig.1).

On disposera de deux variables, **prem** et **dern**, qui contiennent des pointeurs vers la première et la dernière ligne du texte dans l'éditeur.

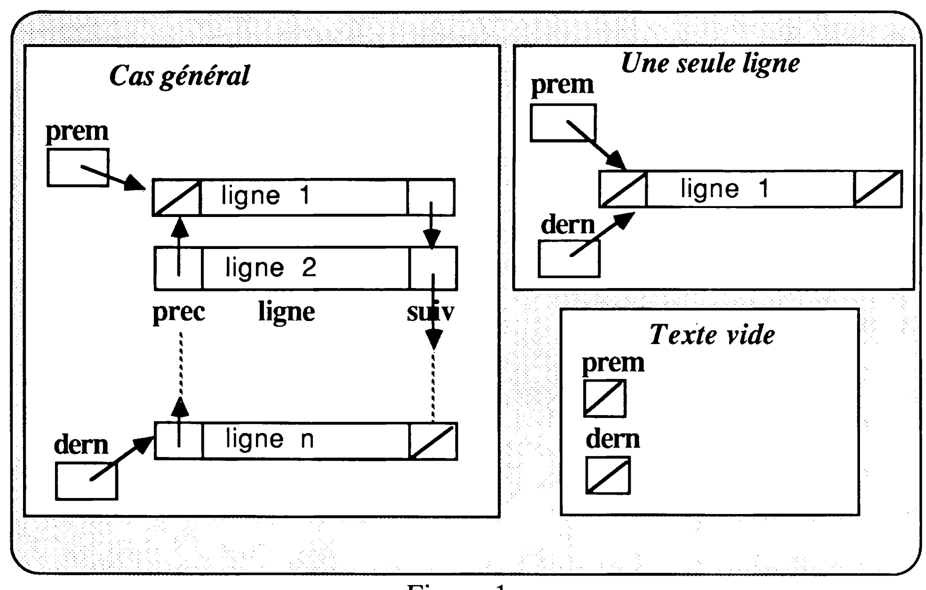


Figure 1

On notera qu'un **texte non vide** vérifiera toujours **prem ≠ nil** et **dern ≠ nil**.

Par ailleurs, on introduira la notion de bloc : un bloc sera formé de lignes consécutives dans le texte, il sera supposé toujours non vide (composé d'au moins une ligne), et sera repéré par les adresses de sa première ligne **db** et de sa dernière ligne **fb** (fig. 2). On notera le bloc : **(db,fb)**, et les relations suivantes seront toujours vérifiées :

$$\mathbf{db} \neq \mathbf{nil},\ \mathbf{fb} \neq \mathbf{nil},$$

$$\mathbf{db} \in \mathbf{prem}^{+} \text{ et } \mathbf{fb} \in \mathbf{db}^{+}.$$

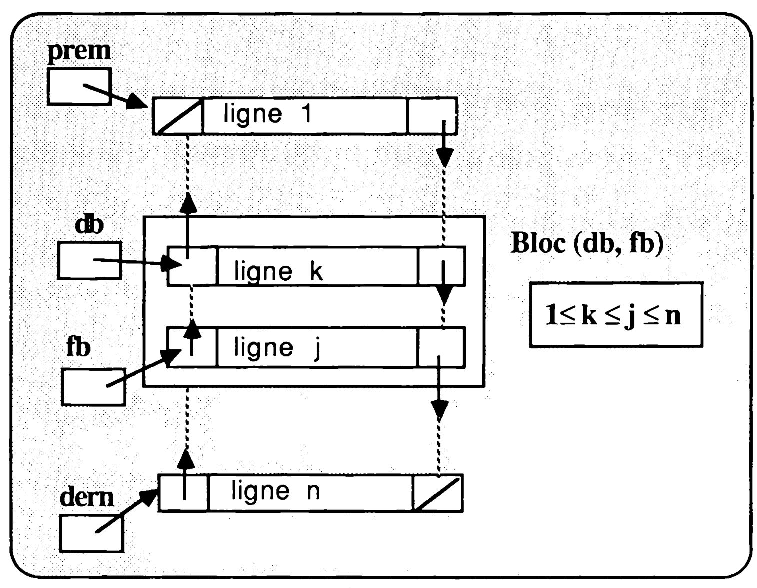


Figure 2

1.1. Écrire la procédure qui initialise le contenu de l'éditeur avec des pointeurs sur un texte vide :
procédure init (r prem, dern : pointeur) ;
spécification { } => {création des pointeurs **prem** et **dern** sur un texte vide}

1.2. Écrire la procédure qui insère une ligne à la fin du texte (initialement vide ou non), et met à jour les pointeurs :
procédure inserfin (dr prem, dern : pointeur ; d lign : ch80) ;
spécification { } => {**lign** a été inséré à la fin du texte}

1.3. Écrire la procédure qui insère une ligne **avant** une ligne d'adresse **pos**. On supposera, pour simplifier, que **pos** est bien l'adresse d'une ligne du texte.
procédure inslignavant (dr prem : pointeur; d pos : pointeur; d lign : ch80);
spécification {texte non vide, $\mathbf{pos}^+ \in \mathbf{prem}^+$} =>
{**lign** a été inséré dans $\mathbf{prem}^+$, avant la ligne d'adresse **pos**}

1.4. Écrire la procédure qui insère une ligne **après** une ligne d'adresse **pos**.
procédure inslignaprès (dr dern : pointeur; d pos : pointeur; d lign : ch80);
spécification {texte non vide, $\mathbf{pos}^+ \in \mathbf{dern}^-$} =>
{**lign** a été inséré dans $\mathbf{dern}^-$, après la ligne d'adresse **pos**}

1.5. Écrire la procédure qui détruit une ligne d'adresse donnée **pos**. On supposera encore que **pos** est bien l'adresse d'une ligne du texte.
procédure détruiligne (dr prem, dern : pointeur; d pos : pointeur);
spécification {texte non vide, **pos**$^{+}$∈**prem**$^{+}$} =>
{la ligne d'adresse **pos** a été détruite}

1.6. Écrire la procédure qui déplace un bloc de lignes **(db, fb)** en la réinsérant avant une ligne d'adresse donnée **dest**. On supposera que toutes les données sont correctes, et que **dest** est à l'extérieur du bloc.
procédure déplacebloc (dr prem, dern : pointeur; d db, fb, dest : pointeur);
spécification {texte non vide, **dest**$^{+}$∈**prem**$^{+}$, **dest** ∉ bloc (**db** , **fb**)} =>
{le bloc a été déplacé avant la ligne d'adresse **dest**}

1.7. Écrire la procédure qui détruit le bloc **(db,fb)**, en rendant l'espace occupé à la mémoire :
procédure détruibloc (dr prem, dern : pointeur; d db, fb: pointeur);
spécification {texte non vide} => {le bloc **(db, fb)** a été détruit}

1.8. Écrire la procédure qui recopie un bloc de lignes **(db, fb)** en insérant une copie de ce bloc avant une ligne d'adresse donnée **dest**. On supposera que les données sont correctes, et que la recopie ne se fera pas à l'intérieur du bloc. Par contre, il est possible que **dest** soit égal à **db**.
procédure copiebloc (dr prem : pointeur; d db, fb, dest : pointeur);
spécification {texte non vide, **dest**$^{+}$∈**prem**$^{+}$, **dest** ∉ bloc (**db**↑**.suiv** , **fb**)} =>
{le bloc a été recopié avant la ligne d'adresse **dest**}

1.9. Écrire la procédure qui charge dans l'éditeur un texte contenu dans un fichier :
procédure chargefichier (dr f : text ; r prem, dern : pointeur) ;
spécification { } => {création d'un texte formé de toutes les lignes de **f**}

1.10. Écrire la procédure qui range le contenu de l'éditeur dans un fichier :
procédure rangefichier (dr f : text ; d prem : pointeur);
spécification { } => {écriture des lignes de **prem**$^{+}$ dans **f**}

2. DEUXIEME VERSION DE L'ÉDITEUR : accès au texte par un seul pointeur, sur une cellule "sentinelle", avec structure en anneau (fig. 3).

On disposera d'une variable **list** qui contiendra toujours l'adresse d'une cellule dite "sentinelle", placée en début du texte, et dont le contenu n'est pas pris en compte. On évite ainsi tous les traitements de cas particuliers relatifs à l'insertion et à la suppression de la première ligne.

Pour obtenir la même facilité pour les traitements de la dernière ligne, on pourrait envisager une deuxième cellule sentinelle à la fin du texte. Il est plus simple d'opter pour une structure en anneau, de telle sorte que la sentinelle soit placée à la fois au début et à la fin du texte. Il suffit pour cela que le pointeur **prec** de la sentinelle contienne l'adresse de la dernière ligne du texte, et que le pointeur **suiv** de la dernière ligne contienne l'adresse de la sentinelle.

Un texte vide sera représenté dans cette structure par la sentinelle seule, dont les pointeurs **prec** et **suiv** pointent sur elle-même. Il vérifiera donc :

list^.prec := list et **list ^.suiv := list.**

Dans le cas général, le texte commencera en **liste^.suiv**. On conviendra que **liste^.suiv+** représente l'ensemble du texte, avec arrêt sur la dernière ligne (**liste^.prec**).

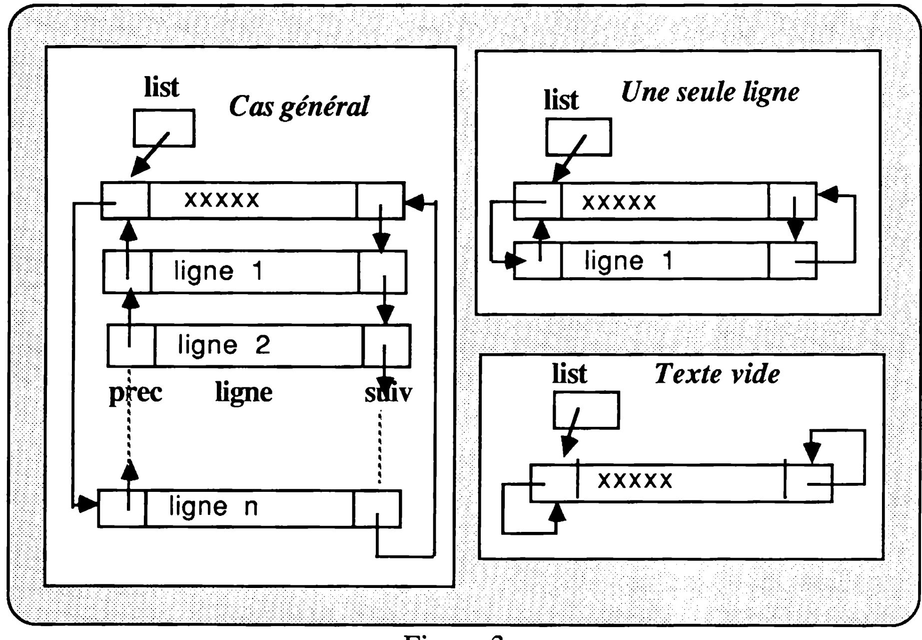


Figure 3

On verra que cette structure, d'apparence plus complexe que la précédente, conduit à des algorithmes bien plus simples.

2.1.
procédure init (r list : pointeur) ;
spécification { } => {création du pointeur **list** sur un texte vide}

2.2.
procédure inserfin (d list : pointeur ; d lign : ch80) ;
spécification { } => {**lign** a été inséré à la fin du texte}

2.3.
procédure inslignavant (d pos : pointeur; d lign : ch80);
spécification {texte non vide, $\mathbf{pos^+} \in$ **list↑.suiv+**} =>
{**lign** a été inséré avant la ligne d'adresse **pos**}

2.4.
procédure inslignaprès (d pos : pointeur; d lign : ch80);
spécification {texte non vide, $\mathbf{pos^+} \in$ **list↑.suiv+**} =>
{**lign** a été inséré après la ligne d'adresse **pos**}

2.5.
procédure détruiligne (d pos : pointeur);
spécification {texte non vide, $\mathbf{pos^+} \in$ **list↑.suiv+**} =>
{la ligne d'adresse **pos** a été détruite}

Pour les procédures suivantes, qui traitent les blocs, on conviendra que la définition d'un bloc est la même que dans la première structure, le bloc ne contient pas de cellule sentinelle. Le bloc **(db, fb)** vérifiera donc :

db $\neq$ **nil**, **fb** $\neq$ **nil**, **db** $\in$ list ↑. suiv+ et **fb** $\in$ **db+**.

2.6.
procédure déplacebloc (d db, fb, dest : pointeur);
spécification {texte non vide, $\mathbf{dest^+} \in$ **list↑.suiv+**, **dest** $\notin$ bloc (**db** , **fb**)} =>
{le bloc a été déplacé avant la ligne d'adresse **dest**}

2.7.
procédure détruibloc (d db, fb: pointeur);
spécification {texte non vide} => {le bloc **(db, fb)** a été détruit}

2.8.
procédure copiebloc (d db, fb, dest : pointeur);
spécification {texte non vide, $\mathbf{dest^+} \in$ **list↑.suiv+**, **dest** $\notin$ bloc (**db↑.suiv** , **fb**)}
=> {le bloc **(db, fb)** a été recopié avant la ligne d'adresse **dest**}

2.9.
procédure chargefichier (dr f : text ; r list : pointeur) ;
spécification { } => {création d'un texte formé de toutes les lignes de **f**}

2.10.
procédure rangefichier (dr f : text ; d list : pointeur);
spécification { } => {écriture des lignes de **list↑.suiv+** dans **f**}

Solutions proposées

1. PREMIERE VERSION DE L'ÉDITEUR

1.1.

procédure init (r prem, dern : pointeur) ;
spécification { } => {création des pointeurs **prem** et **dern** sur un texte vide}
début
  prem := nil; dern := nil
fin;

1.2.

procédure inserfin (dr prem, dern : pointeur ; d lign : ch80) ;
spécification { } => {**lign** a été inséré à la fin du texte}
var x : pointeur ;
début
  nouveau (x) ; x ↑ . ligne := lign ;
  x ↑ . prec := dern ;
  x ↑ . suiv := nil ;
  si prem = nil *{texte vide}* **alors**
    prem := x
  sinon *{prem ≠ nil, donc dern ≠ nil}*
    dern ↑. suiv := x ;
  dern := x
fin;

Dans la figure 4, comme dans les suivantes, les anciens pointeurs modifiés sont grisés, alors que les nouveaux pointeurs sont figurés en gras.

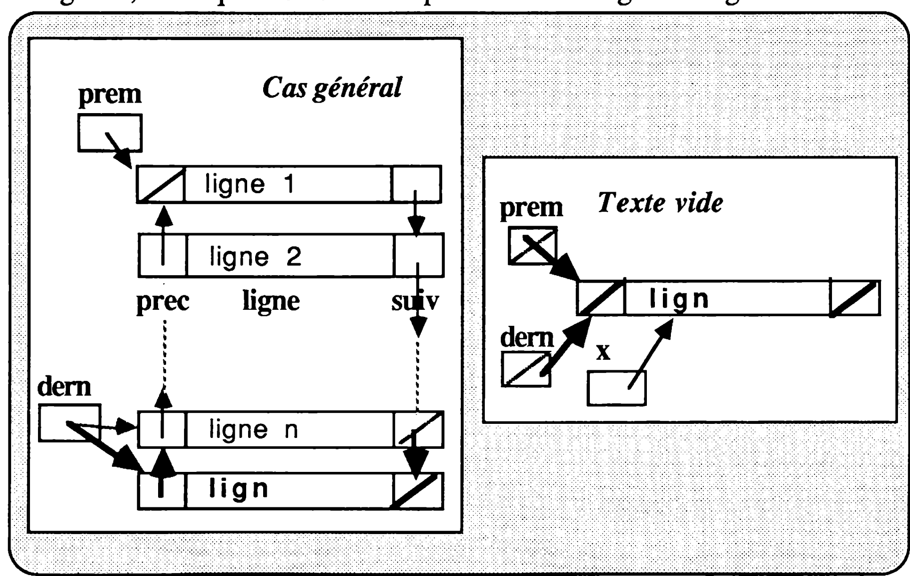


Figure 4

1.3.

procédure inslignavant (dr prem : pointeur; d pos : pointeur; d lign : ch80);
spécification {texte non vide, **pos**$^+$∈ **prem**$^+$} =>
{**lign** a été inséré dans **prem**$^+$, avant la ligne d'adresse **pos**}
var x : pointeur;
début
  nouveau (x); x↑.ligne := lign;
  x↑.suiv := pos;
  x↑.prec := pos↑.prec; pos↑.prec := x;
  si pos = prem *{insertion en tête}* **alors**
    prem := x
  sinon *{cas général}*
    x↑.prec↑.suiv := x
fin;

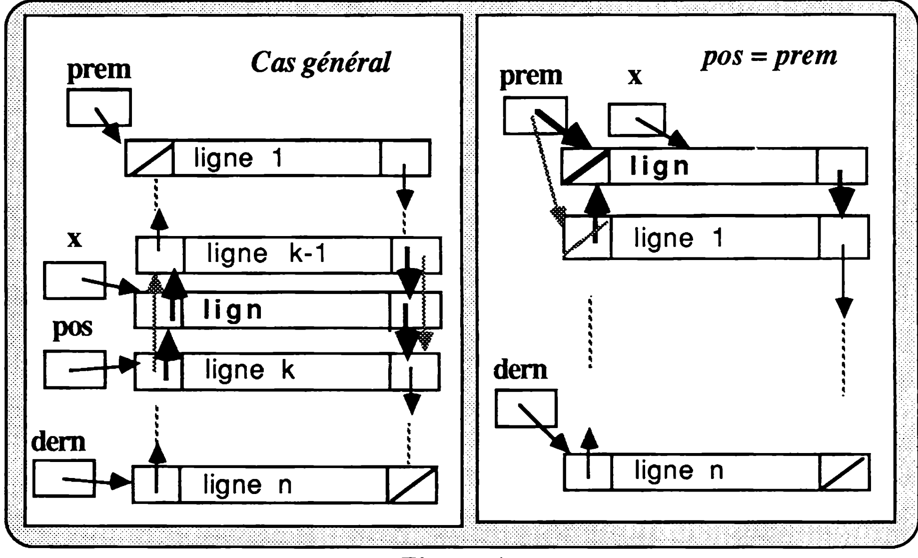


Figure 5

1.4.

procédure inslignaprès (dr dern :pointeur; d pos : pointeur; d lign : ch80);
spécification {texte non vide, **pos**$^+$∈ **dern**$^-$} =>
{**lign** a été inséré dans **dern**$^-$, après la ligne d'adresse **pos**}
var x : pointeur;
début
  nouveau (x); x↑.ligne := lign;
  x↑.prec := pos;
  x↑.suiv := pos↑.suiv; pos↑.suiv := x;

si pos = dern *{insertion en fin de texte}* **alors**
dern := x
sinon *{cas général}*
x↑.suiv↑.prec := x
fin;

Les schémas sont exactement les symétriques de ceux de la question précédente.

1.5. On traitera tous les cas particuliers (liste qui devient vide, suppression au début ou à la fin) d'abord, afin de ne pas avoir de pointeur nil dans le cas général (Fig. 6) .

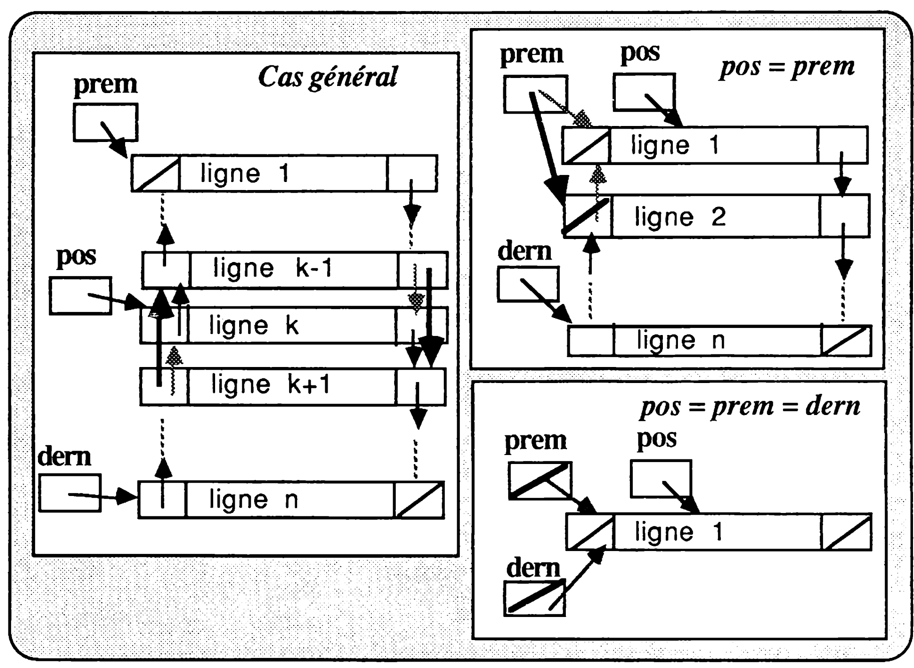


Figure 6

procédure détruiligne (dr prem, dern : pointeur; d pos : pointeur);
spécification {texte non vide, **pos**+∈**prem**+} =>
{la ligne d'adresse **pos** a été détruite}
début
si prem = dern *{la liste ne comporte qu'une ligne}* **alors**
init (prem, dern)
sinon
si pos = prem *{suppression en tête}* **alors**
début
prem := pos↑.suiv; prem↑.prec := nil
fin

```
        sinon
            si pos = dern {suppression à la fin} alors
            début
                dern := pos↑.prec; dern↑.suiv := nil
            fin
            sinon {cas général}
            début
                pos↑.prec↑.suiv := pos↑.suiv;
                pos↑.suiv↑.prec := pos↑.prec;
            fin;
    {rendre la cellule à la mémoire}
    laisser (pos)
fin;
```

1.6. Il faut, dans un premier temps, "couper" le bloc, c'est-à-dire raccorder la ligne au-dessus du bloc à celle qui est au dessous du bloc, à l'aide des deux pointeurs **prec** et **suiv**. On prévoira des cas particuliers si **db** est la première ligne ou bien si **fb** est la dernière ligne. Si ces deux conditions sont vérifiées en même temps, le bloc est alors le texte entier, et son déplacement n'a pas de sens.
On pourra ensuite réinsérer ("coller") le bloc avant **dest**, de la même manière que dans la procédure **inslignavant**.

```
procédure déplacebloc (dr prem, dern : pointeur; d db, fb, dest : pointeur);
spécification {texte non vide, dest+∈prem+, dest ∉ bloc (db , fb)} =>
                        {le bloc a été déplacé avant la ligne d'adresse dest}
début
    {couper le bloc}
    si (db ≠ prem) ou (fb ≠ dern) alors
        si db = prem {bloc en tête} alors
        début {fb ≠nil et fb≠ dern}
            prem := fb↑.suiv;
            prem↑.prec := nil
        fin
        sinon
            si fb = dern {bloc à la fin} alors
            début {db≠nil et db≠prem}
                dern := db↑.prec;
                dern↑.suiv := nil
            fin
            sinon {cas général}
            début
                db↑.prec↑.suiv := fb↑.suiv;
                fb↑.suiv↑.prec := db↑.prec;
            fin;
```

```
    {recoller le bloc avant dest}
    fb↑.suiv := dest;
    db↑.prec := dest↑.prec;
    dest↑.prec := fb;
    si dest = prem  {insertion en tête} alors
        prem := db
    sinon {cas général}
        db↑.prec↑.suiv := db
fin;
```

1.7. Il faut ici d'abord "couper" le bloc, exactement comme dans la procédure de déplacement de bloc, puis rendre chacune de ses lignes à la mémoire. Pour cette procédure, il faut aussi prévoir le cas où le bloc est le texte entier.

```
procédure détruibloc (dr prem, dern : pointeur; d db, fb: pointeur);
spécification {texte non vide} => {le bloc (db, fb) a été détruit}
var x : pointeur ;
début
    {couper le bloc}
    si (db = prem) et (fb = dern)  {destruction du texte entier} alors
        init (prem, dern)
    sinon
        si db = prem  {bloc en tête} alors
        début  {fb ≠ dern et  fb ≠nil}
            prem := fb↑.suiv;
            prem↑.prec := nil
        fin
        sinon
            si fb = dern {bloc à la fin} alors
            début {db ≠prem et db≠nil}
                dern := db↑.prec;
                dern↑.suiv := nil
            fin
            sinon {cas général}
            début
                db↑.prec↑.suiv := fb↑.suiv;
                fb↑.suiv↑.prec := db↑.prec;
            fin;

    {rendre  les cellules à la mémoire}
    tantque db ≠ fb↑.suiv  faire
    début
        x  :=  db ;  db := db↑.suiv;  laisser(x)
    fin
fin;
```

1.8. Pour cette procédure, il suffit d'insérer avant **dest** une copie de chacune des lignes du bloc.

```
procédure copiebloc (dr prem : pointeur; d db, fb, dest : pointeur);
spécification {texte non vide, dest+∈prem+, dest ∉ bloc (db , fb)} =>
                              {le bloc a été recopié avant la ligne d'adresse dest}
début
    tantque db ≠ fb↑.suiv faire
    début
        inslignavant (prem, dest,  db↑.ligne);
        db := db↑.suiv
    fin
fin;
```

1.9. On initialise un texte vide, puis on insère successivement à la fin de ce texte chacune des lignes lues dans le fichier.

```
procédure chargefichier (dr f : text ; r prem, dern : pointeur) ;
spécification { } => {création d'un texte formé de toutes les lignes de f}
var  l:pointeur;
     lign:ch80;
début
    {création d'un texte vide}
    init (prem, dern) ;
    {insertion des lignes de f}
    relire (f);
    tantque non fdf (f) faire
    début
        lireln (f, lign)
        inserfin (prem, dern, lign);
    fin
fin;
```

1.10. Un simple parcours du texte, à partir de l'adresse **prem,** permet de copier chacune des lignes dans le fichier.

```
procédure rangefichier (dr f  : text ; d prem : pointeur);
spécification { } => {écriture des lignes de prem+ dans f}
var l : pointeur;
début
    récrire (f);
    tantque prem ≠ nil { prem :paramètre  "donnée",  donc protégé} faire
    début
        écrireln (f, prem↑.ligne);
        prem := prem↑.suiv
    fin;
fin;
```

2. DEUXIEME VERSION DE L'ÉDITEUR

2.1. Rappelons que le texte vide est formé ici d'une cellule "sentinelle" qui pointe sur elle-même dans les deux sens.

procédure init (r list : pointeur) ;
spécification { } => {création du pointeur **list** sur un texte vide}
début
    nouveau (list);
    list↑.prec := list;
    list↑.suiv := list
fin;

2.2. Schéma de la procédure **inserfin :**

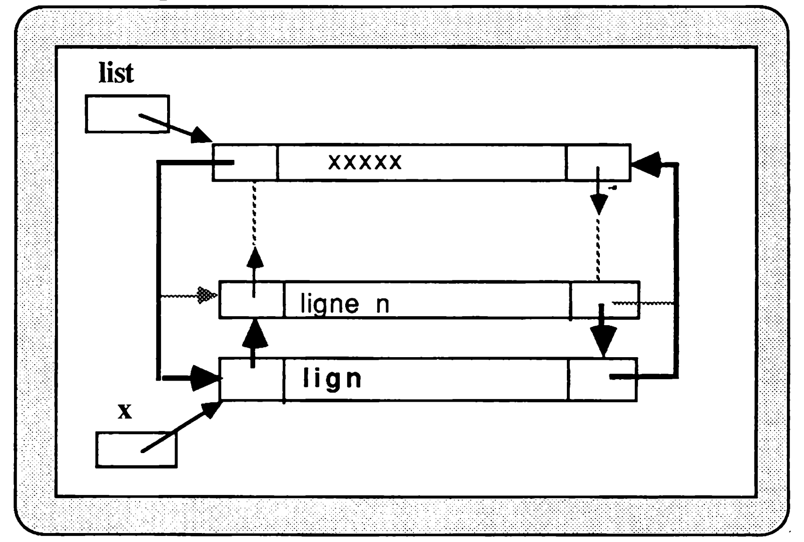


Figure 7

procédure inserfin (d list : pointeur ; d lign : ch80) ;
spécification { } => {**lign** a été inséré à la fin du texte}
var x : pointeur ;
début
    nouveau (x) ; x ↑ . ligne := lign ;
    x ↑ . suiv := list ;
    x ↑ . prec := list ↑. prec ;
    list ↑. prec ↑. suiv := x ;
    list ↑. prec := x
fin;

2.3. Dans toutes les procédures qui suivent, il est inutile de prévoir des cas particuliers pour la première et la dernière ligne ; on est donc ramené au "cas général" des procédures de la première version de l'éditeur.

procédure inslignavant (d pos : pointeur; d lign : ch80);
spécification {texte non vide, **pos⁺∈ list↑.suiv⁺**} =>
{**lign** a été inséré avant la ligne d'adresse **pos**}
var x : pointeur;
début
nouveau (x);
x↑.ligne := lign;
x↑.suiv := pos;
x↑.prec := pos↑.prec;
pos↑.prec := x;
x↑.prec↑.suiv := x
fin;

2.4.
procédure inslignaprès (d pos : pointeur; d lign : ch80);
spécification {texte non vide, **pos⁺∈ list↑.suiv⁺**} =>
{**lign** a été inséré après la ligne d'adresse **pos**}
var x : pointeur;
début
nouveau (x);
x↑.ligne := lign;
x↑.prec := pos;
x↑.suiv := pos↑.suiv;
pos↑.suiv := x;
x↑.suiv↑.prec := x
fin;

2.5.
procédure détruiligne (d pos : pointeur);
spécification {texte non vide, **pos⁺∈ list↑.suiv⁺**} =>
{la ligne d'adresse **pos** a été détruite}
début
pos↑.prec↑.suiv := pos↑.suiv;
pos↑.suiv↑.prec := pos↑.prec;
laisser (pos)
fin;

2.6.
procédure déplacebloc (d db, fb, dest : pointeur);
spécification {texte non vide, **dest⁺∈ list↑.suiv⁺, dest** ∉ bloc (db , **fb**)} =>
{le bloc a été déplacé avant la ligne d'adresse **dest**}
début
{couper le bloc}
db↑.prec↑.suiv := fb↑.suiv;
fb↑.suiv↑.prec := db↑.prec;

```
    {recoller le bloc avant dest}
    fb↑.suiv := dest;
    db↑.prec := dest↑.prec;
    dest↑.prec := fb;
    db↑.prec↑.suiv := db
fin;
```

2.7. On peut remarquer que la variable auxiliaire **x** de 1.7 devient inutile : en effet, même si **fb** est la dernière ligne du bloc, **fb↑.suiv ≠ nil**, donc **db↑.prec** est toujours défini.

```
procédure detruibloc (d db, fb: pointeur);
spécification {texte non vide} => {le bloc (db, fb) a été détruit}
début
    {couper  le bloc}
    db↑.prec↑.suiv := fb↑.suiv;
    fb↑.suiv↑.prec := db↑.prec;

     {rendre mémoire}
    tantque db ≠ fb↑.suiv faire
    début
        db := db↑.suiv;
        laisser (db^.prec)
    fin
fin;
```

2.8.

```
procédure copiebloc (d db, fb, dest : pointeur);
spécification {texte non vide, dest+∈list↑.suiv+, dest ∉ bloc (db↑.suiv , fb)}
                => {le bloc (db, fb) a été recopié avant la ligne d'adresse dest}
var  dbcop, fbcop : pointeur;
début
    tantque db ≠ fb↑.suiv faire
    début
        inslignavant (dest , db↑.ligne);
        db := db↑.suiv
    fin
fin;
```

2.9.

```
procédure chargefichier (dr f : text ; r list : pointeur) ;
spécification { } => {création d'un texte formé de toutes les lignes de f}
var  l : pointeur;
    lign : ch80;
```

```
début{création d'un texte vide}
      init (list) ;
     {insertion des lignes de f}
     relire (f);
     tantque non fdf (f) faire
     début
          lireln (f, lign) ;
          inserfin (list, lign)
     fin
fin;
```

2.10.

```
procédure rangefichier (dr f : text ; d list : pointeur);
spécification { } => {écriture des lignes de list↑.suiv+ dans f}
var l : pointeur;
début
     récrire (f);
     l := list↑.suiv;
     tantque l ≠ list  faire  {structure en anneau : la fin est atteinte lorsqu'on
     début                                         retrouve le début}
          écrireln (f, l↑.ligne);
          l := l↑.suiv
      fin
fin;
```

6

LES TABLES

6.1. Jeux olympiques

(MATRICES CREUSES)

Énoncé

Soit à organiser les inscriptions des athlètes pour les Jeux Olympiques. On supposera qu'il y a au maximum 200 épreuves et 150 pays.

Les **nbpays** pays représentés sont répertoriés, dans l'ordre alphabétique, dans une table **tabpays**. De même, les noms des **nbépreuve** épreuves prévues sont répertoriés, dans l'ordre alphabétique, dans une table **tabépreuve**.

Chaque pays peut présenter de 0 à 10 athlètes dans chaque épreuve. Les noms des athlètes figurent tous dans une matrice (table à deux dimensions) **tabjo**, dont les indices de ligne sont les numéros d'épreuve, les indices de colonne sont les numéros de pays, et chaque élément est une structure de type *concurrents*, comportant le nombre d'athlètes présentés par un pays dans une épreuve, suivi d'une table non triée de dix noms. La table pourra être vide ou partiellement remplie, si le nombre d'athlètes présentés par le pays est inférieur à dix.

```
const maxpays      =  150;
      maxépreuves  =  200;
      maxathlètes  =  10;
type  nom          =  chaîne 25 ;
      concurrents  =  structure
                           nombre    : 0..maxathlètes;
                           tabathlètes : tableau [1..maxathlètes ] de nom
                      fin;
```

var {toutes les variables déclarées ci-dessous sont des variables globales; elles pourront être utilisées, et éventuellement modifiées, dans toutes les procédures}

nbpays,nbépreuve : entier;
tabpays : tableau [1..maxpays] de nom;
tabépreuve : tableau [1..maxépreuves] de nom;
tabjo : tableau [1..maxépreuves, 1..maxpays] de concurrents;

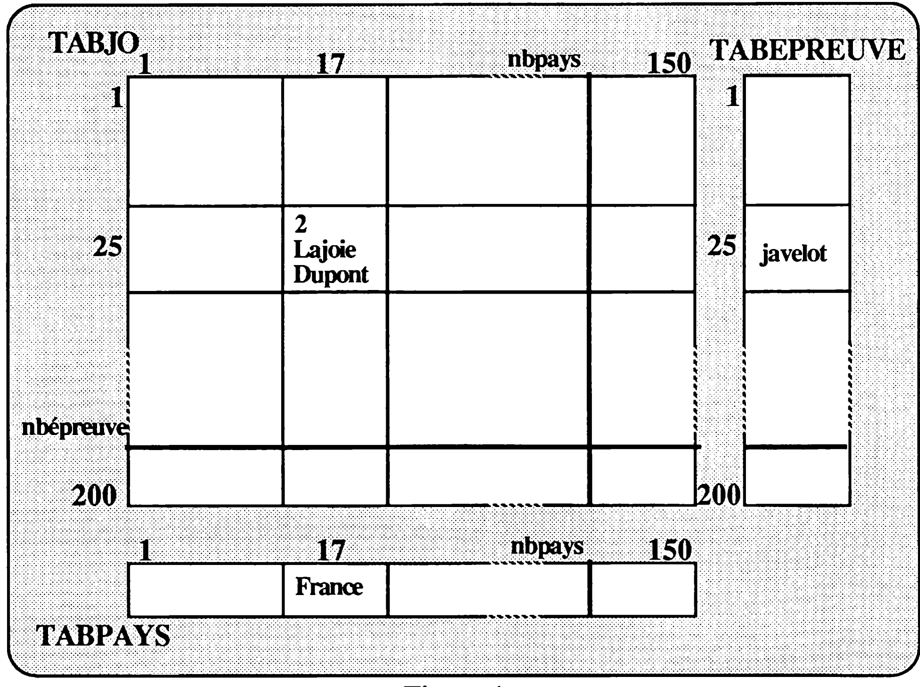


Figure 1

Exemple: La FRANCE, pays n° 17, présente deux concurrents (LAJOIE et DUPONT) dans l'épreuve n° 25, le JAVELOT (cf Fig. 1).

1.1. Écrire, **en utilisant une méthode dichotomique**, la
fonction numpays (d pays : nom) : entier;
spécification { } => {(**numpays** = i , i ∈ [1..nbpays] , tabpays [i] = **pays**)
∨ (**numpays** = 0 , **pays** ∉ tabpays [1..nbpays])}

a) Raisonnement par récurrence
b) Algorithme

1.2. Calculer la taille, en octets, de la matrice **tabjo,** en supposant que le type **chaîne25** est représenté sur 26 octets.

2 . On suppose maintenant que :
- nbpays=100 (nombre de pays effectivement présents) ;
- le nombre **moyen** d'épeuves auxquelles un pays participe est de 50 ;
- pour chaque épreuve à laquelle un pays participe, ce pays présente **en moyenne** trois athlètes .

On modifie alors la structure de la manière suivante : chaque élément de tabjo est un pointeur sur une liste chaînée, non triée, des noms des athlètes qui participent à l'épreuve pour le pays (ou nil s'il n'y en a pas) .

Le type **concurrents** est donc remplacé par les types :

```
pointeur  =   ↑athlète;
athlète   =   structure
                  nomathlète : nom;
                  suivant : pointeur;
              fin;
```

et la variable **tabjo** devient :

tabjo : tableau [1..maxépreuves, 1..maxpays] de pointeur ;

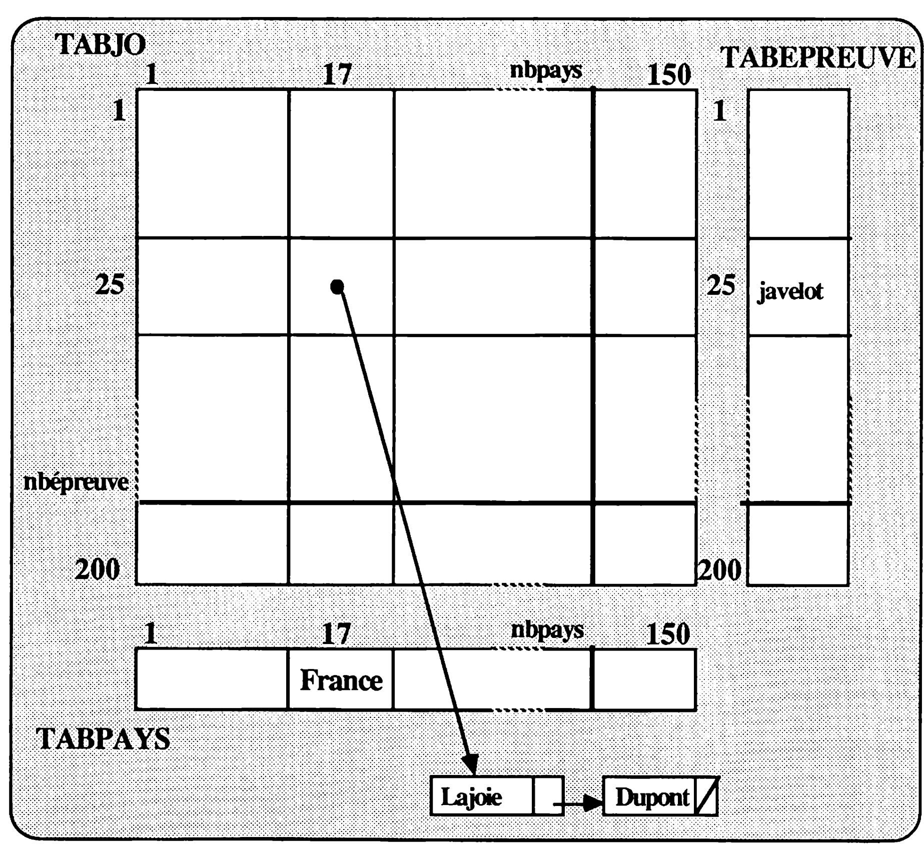


Figure 2

2.1. Calculer la taille **moyenne** en octets de la nouvelle matrice **tabjo** et des listes d'athlètes (en supposant que la taille d'un pointeur est de 4 octets).

2.2. Écrire la procédure qui permet d'insérer un nouvel athlète, défini par son **nom**, son numéro de pays **nump** et son numéro d'épreuve **nume** :
procédure inserath (d nomath : nom ; d nump , nume : entier) ;
La procédure ne devra effectuer l'insertion que si les deux conditions suivantes sont vérifiées pour ce pays et cette épreuve : il y a moins de **maxathlètes** athlètes déjà inscrits, et **nomath** n'y figure pas.

2.3. Écrire la procédure qui permet d'imprimer les noms de tous les athlètes d'un pays donné, en les classant par épreuve :
procédure listathpays (d pays : nom) ;
On devra imprimer le nom de chaque épreuve pour laquelle au moins un athlète du pays est inscrit, suivi de la liste (non triée) des noms des athlètes. Si le pays n'est pas connu, on imprimera un message d'erreur.

2.4. Écrire une fonction
fonction sansath (d ne : entier) : booléen ;
qui ne délivre vrai que si la ligne **ne** comporte uniquement des **nil**, c'est-à-dire si aucun athlète n'est inscrit à l'épreuve numéro **ne**.

2.5. Écrire une procédure
procédure listjo ;
qui affiche le contenu complet de la matrice **tabjo**, épreuve par épreuve : pour chaque épreuve à laquelle un athlète au moins est inscrit, on affiche le nom de l'épreuve, puis, pour chaque pays ayant présenté des concurrents à cette épreuve, le nom du pays et les noms des inscrits.

2.6. Un pays annonce qu'il a décidé, pour des raisons budgétaires ou politiques, de ne plus participer aux Jeux Olympiques. Écrire la procédure qui permet de le supprimer des structures **tabjo** et **tabpays** :
procédure suppays (d pays : nom) ;
On pensera également à libérer l'espace mémoire occupé par les listes.

2.7. Écrire une procédure
procédure supépreuve (d ne : entier) ;
qui supprime l'épreuve de numéro **ne** dans les structures **tabjo** et **tabépreuve**.

2.8. Écrire une procédure
procédure suplignesvides;
qui supprime toutes les épreuves pour lesquelles aucune inscription n'a été enregistrée.

2.9. On dispose d'un fichier de type *text* qui contient les informations suivantes :
- chaque nom de pays est suivi d'une ou plusieurs listes d'épreuves, puis d'un astérisque '*' , qui indique que la ligne suivante, si elle existe, sera un nom de pays.
- chaque liste d'épreuves est formée d'un nom d'épreuve, suivi d'un nombre entier (nombre d'athlètes n) puis des n noms correspondants.

La fin de fichier ne peut donc apparaître qu'après un astérisque '*'.
Le fichier n'est pas trié sur les noms de pays ni sur les noms d'épreuves. Ces noms peuvent figurer dans un ordre quelconque, et même éventuellement plusieurs fois. **N.B. On suppose que le fichier ne comporte aucune erreur.**

Exemple de partie de fichier :

```
.....
....
MEXIQUE
PERCHE
4
LOPEZ
GOMEZ
RAMIREZ
VELEZ
100METRES
1
SUAREZ
*
FRANCE
JAVELOT
2
LAJOIE
DUPONT
*
.....
.....
```

Écrire la procédure qui range tous ces renseignements dans la matrice **tabjo:**
procédure rangejo (dr fichjo : text) ;.

Remarques :
- Les tables **tabpays** (avec **nbpays** connus) et **tabépreuve** (avec **nbépreuve** épreuves) sont supposées être déjà créées . La matrice **tabjo** a été initialisée avec nil dans tous ses éléments, et éventuellement partiellement remplie.
- On supposera qu'il existe une fonction d'en-tête :

fonction numépreuve (d épreu : nom) : entier ;
qui réalise une opération analogue à celle de la fonction **numpays** mais pour une recherche dans le tableau **tabépreuve**.

3. Pour que les données occupent moins de place encore, on va appliquer la technique de la matrice de booléens (cf Tome 2, p. 161). Pour cela, on utilisera les structures suivantes qui remplaceront **tabjo** (cf Fig. 3) :

const nbptr = 6000;
var tabjobool : tableau [1..maxépreuves, 1..maxpays] de booléen ;
vptr : tableau [1..nbptr] de pointeur;

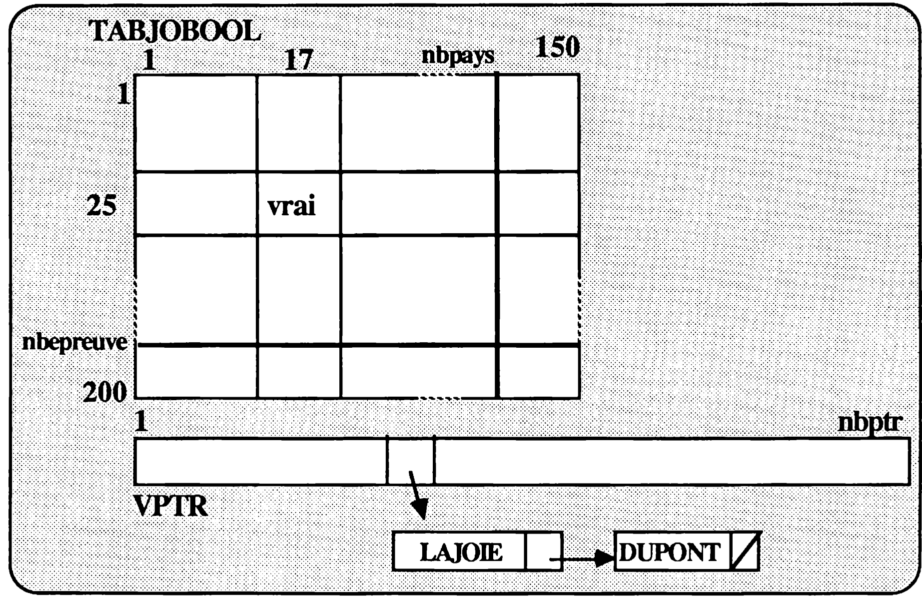


Figure 3

Si un élément de **tabjobool** est **faux**, c'est que le pointeur correspondant est **nil**, on ne le représente pas ; si cet élément est **vrai**, on trouve le pointeur correspondant dans **vptr**.
En gardant les hypothèses proposées au début de la partie 2 (100 pays, 50 épreuves en moyenne), il faudra 5000 pointeurs ; on peut prendre **nbptr** égal à 6000 par précaution.

3.1. Calculer la taille des nouvelles structures, dans les deux hypothèses suivantes :
a) un booléen occupe un octet ;
b) un booléen occupe un élément binaire (bit).

3.2. Écrire la fonction qui permet d'accéder à la valeur d'un pointeur, et qui devra remplacer l'indiçage des procédures écrites ci-dessus :
fonction tabjo (d ne , np : entier) : pointeur;
Cette fonction délivre **nil** si **tabjobool [ne, np]** vaut **faux**; sinon, il faut compter le nombre de **vrai** avant le couple **(ne, np)**, en lui ajoutant 1 pour accéder au bon élément de **vptr**.

Solutions proposées

1.1.a) Le raisonnement est presque le même que celui du cours (cf Tome 1, p. 136).

Hypothèse de récurrence A :

tabpays [1..inf-1] < pays < tabpays [sup+1..nbpays]

∗ inf=sup+1 => tabpays [1..inf-1] < pays < tabpays [inf..nbpays]
  => **numpays=0**
∗ inf≤sup
  soit m=(inf+sup)/2
  ∗∗ tabpays [m] = pays => **numpays=m**
  ∗∗ tabpays [m] < pays : on effectue inf:=m+1 pour retrouver A
  ∗∗ tabpays [m] > pays : on effectue sup:=m-1 pour retrouver A

Itération : on est amené à introduire un booléen pour la cohérence
Initialisation (non optimisée): inf:=1; sup:=nbpays;

b) L'algorithme est alors le suivant :

```
fonction numpays (d pays : nom) : entier;
spécification { } => {(numpays = i , i ∈ [1..nbpays ] , tabpays [ i ] = pays)
                      ∨ (numpays = 0 , pays ∉ tabpays [ 1..nbpays ])}
var  inf, sup, m : entier;
     trouvé : booléen;
début
     numpays := 0 ;
     inf := 1;  sup := nbpays ;
     trouvé := faux ;
     tantque (inf ≤ sup)  et non trouvé faire
     début
          m := (inf+sup) div 2;
          si tabpays [  m ] = pays alors
          début
               trouvé := vrai;
               numpays := m
          fin
          sinon
               si tabpays [  m ] < pays alors  inf := m+1
               sinon  sup := m-1
     fin
fin;
```

1.2. Taille de **concurrents** : le nombre occupe 1 octet (entier < 256), chaque nom occupe 26 octets, soit au total 1 + (26*10) = 261 octets .

Taille de **tabjo** : 150*200*261= 7 830 000 octets ou environ **7.5 Mo** .

2.
2.1. Taille de tabjo = 150*200*4 = 120 000 octets ou environ **118 Ko** .
Nombre moyen de listes = 50*100 = 5000 ;
taille moyenne d'une liste = 3*(26 + 4) = 90 octets ;
d'où : taille des listes = 5000*90 = 450 000 octets ou environ **440 Ko** .

Taille moyenne totale = 118Ko + 440 Ko = **570 Ko** , soit moins du dixième de la taille de la première structure.

2.2. Cette procédure se compose de deux parties
- parcours de la liste d'athlètes correspondant à **nump** et **nume**, en s'arrêtant si **nomath** y figure déjà ou s'il y a déjà **maxathlètes** noms dans la liste ;
- puis, si les conditions sont vérifiées, insertion du nouveau nom en tête de la liste.

```
procédure inserath (d nomath : nom ; d nump , nume : entier) ;
var  a , ath : pointeur ;
     i : entier ;
     stop : booléen ;
début
     a := tabjo [ nume , nump ] ; {a = adresse de la liste}
     stop := faux ;
     i := 0 ; {i = compteur}
     tantque (a≠nil) et non stop faire
     début
          i := i+1 ;
          si (i = maxathlètes) ou (a↑.nomathlète = nomath) alors
               stop := vrai
               {les conditions pour l'insertion ne sont pas vérifiées}
          sinon
               a := a↑.suivant
     fin ;
     si non stop alors
     {les conditions pour l'insertion sont vérifiées}
     début
          {insertion en tête de la liste}
          nouveau (ath) ;
          ath↑.nomathlète := nomath ;
          ath↑.suivant := tabjo [ nume , nump ] ;
          tabjo [ nume , nump ] := ath
     fin ;
fin ;
```

2.3. Il s'agit d'un balayage d'une colonne de la matrice, puis pour chaque élément de la colonne qui correspond à une liste non vide, d'un simple parcours de cette liste.

```
procédure listathpays (d pays  :  nom) ;
var  i , np  :  entier ;
     a  :  pointeur ;
début
     np := numpays (pays) ;
     si np = 0 alors
          écrire ('erreur sur pays')
     sinon
          {pour chaque ligne i , ou épreuve, de la colonne np}
          pour i := 1 haut nbépreuve faire
          début
               a := tabjo [ i , np ]    ;
               si a ≠ nil alors
               début
                    {liste non vide, au moins un athlète inscrit}
                    écrireln ('épreuve  :  ' , tabépreuve [ i ]) ;
                    tantque a ≠ nil faire
                    début
                         écrireln (a↑.nomathlète) ;
                         a := a↑.suivant
                    fin
               fin
          fin
fin ;
```

2.4. La recherche dans une ligne de la matrice est identique à la recherche d'une valeur dans un vecteur. On recherchera ici une adresse de liste différente de **nil** : si elle existe, c'est qu'il y a au moins un athlète inscrit pour ce pays, et on pourra alors arrêter la recherche.

```
fonction sansath (d ne : entier) : booléen ;
var  j  : entier ;
     trouvath   : booléen ;
début
     trouvath := faux ;
     j := 1 ;
     tantque (j≤nbpays) et non trouvath faire
          si tabjo [ ne , j ] ≠ nil alors
               trouvath := vrai
          sinon
               j := j+1 ;
     sansath := non trouvath
fin ;
```

2.5. Dans cette procédure on devra faire une double itération, écrite ici à l'aide de deux boucles "pour" imbriquées : parcours par colonnes pour les épreuves, puis, pour chaque épreuve où au moins un athlète est inscrit, parcours par lignes, c'est-à-dire par pays. On remarquera que la boucle interne est tout à fait analogue à la procédure **listathpays**, en échangeant les rôles des épreuves et des pays.

```
procédure listjo ;
var  i , j  :  entier ;
     a  :  pointeur ;
début
     pour i := 1 haut  nbépreuve faire
          {pour chaque épreuve  de numéro i}
          si  non sansath (i) alors
          début
               écrireln ('épreuve  :  ' , tabépreuve [ i ]) ;
               pour j := 1 haut  nbpays faire
               {pour chaque  pays  de numéro j}
               début
                    a := tabjo [ i , j ]     ;
                    si a ≠ nil alors
                    début
                         {s'il y a un inscrit pour i,j}
                         écrireln ('pays  :  ' , tabpays [ j ]) ;
                         {liste des athlètes pour i et j}
                         tantque a ≠ nil faire
                         début
                              écrireln (a↑.nomathlète) ;
                              a :=  a↑.suivant
                         fin ;
                         écrireln
                    fin
               fin
          fin
fin ;
```

2.6. Il faut effectuer deux suppressions : le pays dans la table des pays, et la colonne complète correspondante dans la matrice. Ces suppressions s'opèrent par décalage vers la gauche des éléments situés "à droite" du pays concerné (cf procédure **tasser**, Tome 1, p. 181). La récupération de l'espace occupé par les listes doit être effectuée avant de perdre l'adresse de ces listes.

```
procédure suppays (d pays  :  nom) ;
var  i , j , np  : entier  ;
     a , b  :  pointeur  ;
```

```
début
    np := numpays (pays) ;
    si np ≠ 0 alors
    début
        nbpays := nbpays-1 ;
        {décalage à gauche de tous les pays}
        pour i := np haut nbpays faire
            tabpays [ i ] := tabpays [ i+1 ]    ;
        {pour chaque ligne :}
        pour i := 1 haut nbépreuve faire
        début {rendre chaque liste de la colonne np}
            a := tabjo [ i , np ]    ;
            tantque a ≠ nil faire
            début
                b := a  ; a := a↑.suivant  ; laisser (b)  ;
            fin  ;
            {puis, décalage à gauche de toutes les colonnes}
            pour j := np haut nbpays faire
                tabjo [ i , j ]   := tabjo [ i , j+1 ]    ;
        fin
    fin
fin ;
```

2.7. Cette procédure est la "symétrique" de la précédente, en échangeant les rôles des lignes et des colonnes, donc des épreuves et des pays.

```
procédure supépreuve (d ne : entier) ;
var     i , j :  entier ;
        a , b :  pointeur ;
début
    nbépreuve := nbépreuve-1 ;
    {décalage vers le haut de toutes les épreuves}
    pour i := ne haut nbépreuve faire
        tabépreuve [ i ]   := tabépreuve [ i+1 ] ;
    {pour chaque colonne :}
    pour j := 1 haut nbpays faire
    début {rendre la liste de la ligne ne et de la colonne j}
        a := tabjo [ ne , j ] ;
        tantque a ≠ nil faire
        début
            b := a  ; a := a↑.suivant  ; laisser (b)
        fin  ;
    {décaler vers le haut, d'une position, toutes les lignes de la colonne j}
        pour i := ne haut nbépreuve faire
            tabjo [ i , j ] := tabjo [ i+1 , j ] ;
    fin
fin ;
```

2.8. Il suffit d'appeler le prédicat de test **sansath** pour chaque ligne et, s'il est vérifié, d'effectuer la suppression de cette ligne par la procédure **supépreuve**.

```
procédure suplignesvides;
var  i : entier ;
début
     pour i := 1 to nbépreuve faire
          si sansath (i) alors  supépreuve(i)
fin ;
```

On peut remarquer que cette méthode n'est pas très efficace. En effet, certaines lignes seront décalées plusieurs fois d'une position, et non une seule fois de plusieurs positions. De plus, on appelle inutilement les instructions de suppression de listes, puisque ces listes sont justement vides. Pour obtenir une meilleure efficacité, on peut s'inspirer de la procédure **supbidon** (Tome 1, p. 182) :

```
procédure suplignesvides;
var  i , j , k :  entier ;
     vide : booléen;
début
     {recherche de la première ligne à supprimer}
     i := 1; vide := faux;
     tantque (i ≤ nbépreuve) et non vide faire
          si sansath (i) alors vide := vrai
          sinon i := i+1;
     {retassemenent si nécessaire}
     j := i +1;
     tantque  j ≤ nbépreuve faire
     début
          si non  sansath (j) {ligne à conserver}  alors
          début
               {remonter la jème ligne en ième position}
               pour k := 1 haut nbpays faire tabjo [ i, k ] := tabjo [ j , k ];
               {remonter la jème épreuve en ième position}
               tabépreuve [ i ] := tabépreuve [ j  ] ;
               i := i+1
          fin;
          j := j +1
     fin;
     nbépreuve := i - 1
fin;
```

2.9. Trois boucles itératives imbriquées permettent de lire et de traiter correctement le fichier ; elles correspondent à la "grammaire" du fichier , qu'on pourrait exprimer ainsi :

<fichier> ::= {<pays>}

*<pays> ::= nom-pays {épreuve} **
*<épreuve> ::= nom-épreuve nb-concurrents {nom-concurrent}**
avec une contrainte sémantique : le nombre d'occurrences de *nom-concurrent* doit être égal à *nb-concurrents.*

```
procédure rangejo (dr fichjo : text) ;
var  nompays, noméprreuve, nomath : nom ;
     np, ne, nb : entier ;
début
     relire (fichjo) ;
     {parcours du fichier}
     tantque non fdf (fichjo) faire
     début
          {pour chaque pays dans le fichier}
          lireln (fichjo, nompays) ;
          np := numpays (nompays) ;
          {pour chaque épreuve de ce pays}
          lireln (fichjo, noméprreuve) ;
          tantque noméprreuve ≠ '*' faire
          début
               ne := numépreuve (noméprreuve) ;
               lireln (fichjo, nb) ;
               {pour chaque concurrent de ce pays et cette épreuve}
               tantque nb ≠ 0 faire
               début
                    {lecture et insertion du nom-concurrent}
                    lireln (fichjo, nomath) ;
                    inserath (nomath, np, ne) ;
                    nb := nb-1
               fin ; {fin des concurrents}
               lireln (fichjo, noméprreuve)
          fin {fin des épreuves}
     fin {fin du fichier, donc des pays}
fin ;
```

3.1.a)
Taille de **tabjo** : 150*200*1= 30 000 octets ou environ **29.3 Ko**,
taille de **vptr** : 6000*4 = 24 000 octets, ou environ **23.5 Ko** ;
soit au total moins de **53Ko,** au lieu des 118 Ko de la structure précédente. Les tables tabpays et tabépreuves sont inchangées.

3.1.b)
Taille de **tabjo** : 150*200*1/8= 3750 octets ,
taille de **vptr** : 6000*4 = 24 000 octets ;
soit au total environ **27 Ko**, les tables tabpays et tabépreuves sont toujours inchangées.

3.2.

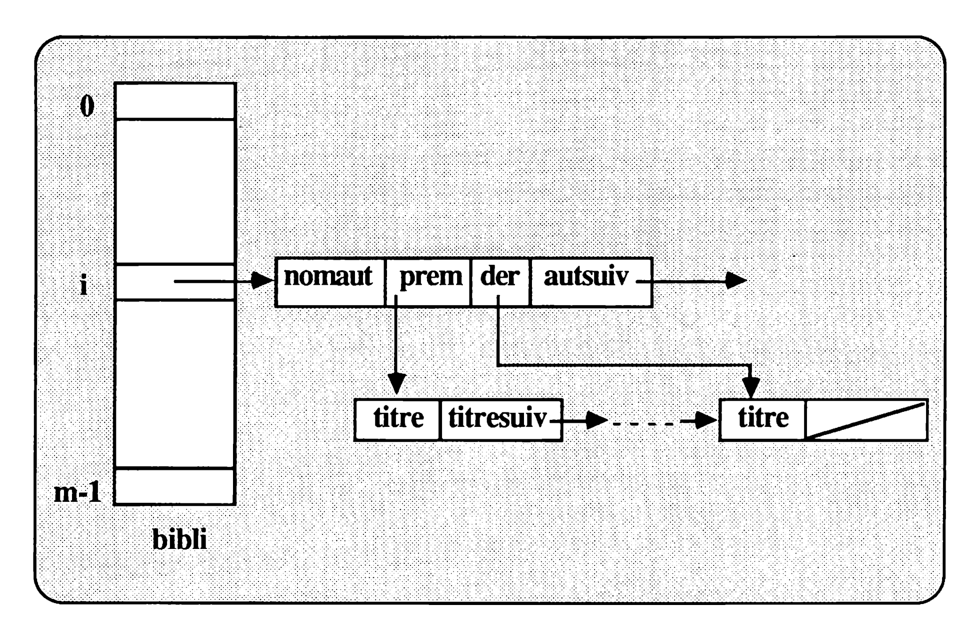
```
fonction tabjo (d ne, np: entier) : pointeur;
var nb : entier;
début
    si non tabjobool [ ne, np]  alors  {liste vide}
        tabjo := nil
    sinon
    début
        nb := 0 ;
        pour  i := 1 haut ne-1 faire
            pour j := 1 haut nbpays faire
                si tabjobool [ i , j]  alors  {liste (i,j) non vide}
                    nb := nb + 1 ;
        {nb contient le nombre de listes non vides avant  la ligne ne}
        pour j := 1 haut np faire
            si tabjobool [ ne , j ] alors
                nb := nb + 1 ;
        {nb contient le nombre de listes non vides jusqu'à (ne,np)}
        tabjo := vptr [ nb ]
    fin
fin;
```

6.2. Gestion d'une bibliothèque

(HASH-CODE CHAINÉ)

Énoncé

On souhaite gérer les livres d'une bibliothèque à l'aide d'une table de hash-code chaînée.
A chaque nom d'auteur correspondra un code compris entre 0 et m-1. Chaque code donne accès à une **liste chaînée triée** de tous les noms d'auteurs ayant le même code.
Pour chaque auteur, on connaît son nom : **nomaut**; on dispose également de deux pointeurs **prem** et **der** sur la liste des livres écrits par l'auteur, **triée par ordre chronologique** : **prem** est le pointeur sur le premier livre écrit par l'auteur alors que **der** pointe sur le dernier livre écrit par l'auteur.
Cette organisation correspond au schéma suivant :



On dispose des déclarations suivantes :

```
const
    m = 64;
type
    ch20   =  chaîne20;
    paut   =  ↑auteur;
    plivre =  ↑livre;
    indtab =  0 .. m-1;
```

```
auteur =  structure
              nomaut : ch20;
              prem, der : plivre;
              autsuiv : paut
          fin;
livre  =  structure
              titre : ch20;
              titresuiv : plivre
          fin;
biblio =  tableau [ indtab] de paut;
```

On supposera que **tous les auteurs ont un nom différent** et que les **titres de tous les livres sont différents.**

On dispose d'une fonction de hash-code intitulée **calculecode** qui calcule le code d'un auteur. L'en-tête de cette fonction est le suivant :
fonction calculecode (d nom : ch20 ; d m : entier) : indtab ;
spécification {**m** $\geq$ **1**} => {**calculecode** = i , 0 $\leq$ i $\leq$ m -1 , i = code du nom d'auteur **nom**}

1. PARCOURS DE LISTES ET COMPTAGES D'ÉLÉMENTS

1.1. Écrire, sous forme itérative puis récursive, une
fonction nbaut (d pa : paut) : entier ;
spécification { } => {**nbaut** = nombre d'auteurs $\in$ **pa**$^+$}

1.2. Écrire une
fonction nbautot (d bibli : biblio ; d m : entier) : entier ;
spécification {**m** $\geq$ **1**} => {**nbautot** = nombre d'auteurs $\in$ **bibli[0..m-1]**}

1.3. Écrire une
fonction pointaut (d bibli : biblio; d m : entier; d nom : ch20) : paut;
spécification {**m≥1 , liste ordonnée des auteurs**} => {(**pointaut** = pointeur sur la cellule contenant le nom d'auteur **nom , nom** $\in$ **bibli[0..m-1]**)
v (**pointaut = nil , nom** $\notin$ **bibli[0..m-1]**)}

1.4. Écrire, sous forme récursive, une
fonction nblivre (d pl : plivre) : entier ;
spécification {**m** $\geq$ **1**} => {**nblivre** = nombre de livres $\in$ **pl**$^+$}

1.5. Écrire une
fonction nblivraut (d bibli : biblio; d m : entier; d nom : ch20) : entier;
spécification {**m** $\geq$ **1,liste ordonnée des auteurs**} =>{(**nom** $\in$ **bibli[0..m-1]**, **nblivraut** = nombre de livres écrits par l'auteur **nom**)
v (**nom** $\notin$ **bibli[0..m-1] , nblivraut = 0**)}

2. INSERTION D'ÉLÉMENTS

2.1. *INSERTION D'UN AUTEUR*

On s'intéresse d'abord à l'insertion d'un auteur n'ayant encore écrit aucun livre. Pour ce faire, on écrira deux procédures auxiliaires dont la première sera utilisée dans la procédure **insaut** et la seconde dans la procédure **insauteur**.

2.1.1. Écrire une
procédure instêteaut (dr pa : paut ; d nom : ch20);
spécification {**nom** $\notin$ **pa+**} => {le nouvel auteur **nom** qui n'a encore écrit aucun livre a été inséré en tête de **pa+**}

2.1.2. En utilisant la procédure **instêteaut,** écrire, sous forme récursive, une
procédure insaut (dr pa : paut ; d nom : ch20 ; r pauteur:paut);
spécification {**pa+ triée**} => {(**nom** $\notin$ **pa+**, le nouvel auteur **nom** qui n'a encore écrit aucun livre a été inséré dans **pa+**, **pa+ triée**)
v (**nom** $\in$ **pa+** , pas d'insertion) ,
pauteur = adresse de la cellule contenant **nom**}

2.1.3. En utilisant la procédure **insaut,** écrire une
procédure insauteur (dr bibli:biblio;d m:entier;d nom:ch20;r pauteur:paut);
spécification {**m** $\geq$ **1**} => {(**nom** $\notin$ **bibli[0..m-1]**, le nouvel auteur **nom** qui n'a encore écrit aucun livre a été inséré dans **bibli[0..m-1]**)
v (**nom** $\in$ **bibli[0..m-1]** , pas d'insertion) ,
pauteur = adresse de la cellule contenant **nom**}

2.2. *INSERTION D'UN LIVRE*

On s'intéresse maintenant à l'insertion d'un livre. Comme précédemment, on écrira deux procédures auxiliaires.

2.2.1. Écrire une
procédure instêtelivre (dr pl : plivre ; d titre : ch20);
spécification { } => {le livre **titre** a été inséré en tête de **pl+**}

2.2.2. En utilisant la procédure **instêtelivre,** écrire une
procédure inslivre (d pa : paut ; d titre : ch20);
spécification {**pa** $\neq$ **nil**} => {le dernier livre **titre** écrit par **pa↑.nomaut** a été inséré dans la liste des livres écrits par **pa↑.nomaut**}
On rappelle que la liste des livres est ordonnée par ordre chronologique. Attention au fait que ce livre peut être le premier que l'auteur ait écrit.

2.2.3. En utilisant les procédures **précédentes,** écrire une
procédure inserlivre (dr bibli : biblio; d m : entier; d nom , titre : ch20);
spécification {**m** $\geq$ **1**} => {le dernier **titre** écrit par **nom** $\in$ **bibli[0..m-1]**}

3. SUPPRESSIONS D'ÉLÉMENTS

3.1. *SUPPRESSION D'UN LIVRE*

On s'intéresse d'abord à la suppression d'**un livre** écrit par un auteur. Pour ce faire, on écrira des procédures ou fonctions auxiliaires permettant de décomposer le problème en sous-problèmes plus simples.

3.1.1. Écrire, sous forme récursive, une
fonction dernier (d pl : plivre) : plivre;
spécification {**pl ≠ nil**} =>{**dernier** = adresse de la dernière cellule de **pl+**}

3.1.2. En utilisant la fonction précédente, écrire une
procédure supptêtelivre (dr prem , der , pl : plivre) ;
spécification {**pl+ non vide**} => {la cellule de tête de **pl+** a été supprimée , **prem** et **der** ont été mis à jour si nécessaire}

3.1.3. En utilisant la fonction précédente, écrire, sous forme récursive, une
procédure suppunlivre (dr prem , der , pl : plivre ; d titre : ch20) ;
spécification {**pl+ ordonnée**} => {le livre **titre** de **pl+** a été supprimé , **prem** et **der** ont été mis à jour si nécessaire}
Cette procédure ne fait rien si **titre** ∉ **prem+**.

3.1.4. En utilisant la procédure précédente, écrire une
procédure supplivreaut (d bibli : biblio; d m : entier ; d nom,titre : ch20);
spécification {**m ≥ 1**} => {le livre **titre** écrit par **nom** ∉ **bibli[0..m-1]** , **prem** et **der** ont été mis à jour si nécessaire}
Cette procédure n'a aucun effet si l'auteur **nom** ∉ **bibli[0..m-1]** ou s'il n'a pas écrit ce livre.

3.2. *SUPPRESSION D'UN AUTEUR*

On s'intéresse maintenant à la suppression d'**un auteur** et de **tous les livres** qu'il a écrits. Pour ce faire, on écrira des procédures auxiliaires permettant de décomposer le problème en sous-problèmes plus simples.

3.2.1. Écrire, sous forme récursive puis itérative, une
procédure supplivres (dr pl : plivre);
spécification { } => {tous les livres de **pl+** ont été supprimés}

3.2.2. Écrire une
procédure supptêteaut (dr pa : paut) ;
spécification {**pa ≠ nil**} => {la cellule de tête de **pa+** a été supprimée}

3.2.3. En utilisant les procédures **supptêteaut** et **supplivres**, écrire, sous forme récursive, une
procédure suppautl (dr pa : paut ; d nom : ch20) ;
spécification {**pa+ triée**}=>{l'auteur **nom** et **tous ses livres** ∉ **pa+**}

Cette procédure n'a aucun effet si l'auteur **nom** ∉ **pa⁺**.

3.2.4. En utilisant les procédures précédentes, écrire une
procédure suppauteur (dr bibli : biblio ; d m : entier; d nom : ch20) ;
spécification {**m** ≥ **1**} => {l'auteur **nom** et **tous ses livres** ∉ **bibli[0..m-1]**}
Cette procédure n'a aucun effet si l'auteur **nom** ∉ **bibli[0..m-1]**.

Solutions proposées

1. PARCOURS DE LISTES ET COMPTAGES D'ÉLÉMENTS

1.1. Version itérative : parcours classique d'une liste (cf Tome 2, p. 23).

```
fonction nbaut (d pa : paut) : entier ;
spécification { } => {nbaut = nombre d'auteurs ∈ pa+}
var nb : entier ;
début
      nb := 0 ;
      tantque pa ≠ nil faire
      début
            nb := nb + 1 ;
            pa := pa↑.autsuiv
      fin;
      nbaut := nb
fin;
```

Version récursive : parcours d'une liste (cf Tome 2, p. 24).

```
fonction nbaut (d pa : paut) : entier ;
spécification { } => {nbaut = nombre d'auteurs ∈ pa+}
début
      si pa = nil alors  nbaut := 0
      sinon  nbaut := 1 + nbaut(pa↑.autsuiv)
fin;
```

1.2. Cette fonction doit appeler, pour tous les éléments de **bibli,** la fonction précédente **nbaut.**

```
fonction nbautot (d bibli : biblio ; d m : entier) : entier ;
spécification {m ≥ 1} => {nbautot = nombre d'auteurs ∈ bibli[0..m-1]}
var i, nbt : entier;
début
      nbt := 0;
      pour i := 0 haut m - 1 faire
            nbt := nbaut(bibli[i]) + nbt;
      nbautot := nbt
fin;
```

1.3. On peut donner deux versions de la fonction pointaut :

Version itérative

il faut calculer le code de **nom**, puis effectuer un parcours associatif d'une liste ordonnée (cf Tome 2, p. 35).

```
fonction pointaut (d bibli : biblio; d m : entier; d nom : ch20) : paut;
spécification {m≥1 , liste ordonnée des auteurs} => {(pointaut = pointeur
        sur la cellule contenant le nom d'auteur nom , nom ∈ bibli[0..m-1])
                                   ∨ (pointaut = nil , nom ∉ bibli[0..m-1])}
var  t : paut;  infer : booléen;
début
    pointaut := nil; {cas où l'auteur est absent}
    infer := vrai;   t := bibli [calculecode(nom, m)];
    {calcul de l'adresse de tête de liste correspondant au code de nom}
    tantque (t ≠ nil) et infer faire
        si t↑.nomaut < nom alors {la liste est ordonnée}
            t := t↑.autsuiv
        sinon
        début  {t↑.nomaut ≥ nom}
            infer := faux;
            si t↑.nomaut = nom alors
                pointaut := t;
        fin
fin;
```

Version récursive

il faut appeler une fonction auxiliaire **pointautr** qui effectue le parcours de la liste ordonnée (cf Tome2, p. 35).

```
fonction pointaut (d bibli : biblio; d m : entier; d nom : ch20) : paut;
spécification {m≥1 , liste ordonnée des auteurs} => {(pointaut = pointeur
        sur la cellule contenant le nom d'auteur nom , nom ∈ bibli[0..m-1])
                                   ∨ (pointaut = nil , nom ∉ bibli[0..m-1])}
var  t : paut;  infer : booléen;
début
    pointaut :=  pointautr (bibli [calculecode(nom, m)] , nom)
fin ;

fonction pointautr (d pa : paut ; d nom : ch20) : paut ;
spécification {pa+ ordonnée} => {(pointautr = pointeur sur la cellule
                        contenant le nom d'auteur nom , nom ∈ pa+)
                                       ∨ (pointautr = nil , nom ∉ pa+)}
début
    si pa = nil alors
        pointautr := nil
    sinon
        si pa↑.nomaut < nom alors
            pointautr := pointautr (pa↑.autsuiv)
```

```
        sinon
            si pa↑.nomaut = nom alors
                pointautr := pa
            sinon {pa↑.nomaut > nom}
                pointautr := nil
fin;
```

1.4. Parcours classique, sous forme récursive, d'une liste (cf tome 2, p. 24).

```
fonction nblivre (d pl : plivre) : entier ;
spécification {m ≥ 1} => {nblivre = nombre de livres ∈ pl+}
début
    si pl = nil alors nblivre := 0
    sinon nblivre := 1 + nblivre(pl↑.titresuiv)
fin;
```

1.5. Dans un premier temps, il faut trouver l'adresse de la cellule contenant **nom :** on fait appel à la fonction **pointaut.** Ensuite, si l'auteur est absent, la fonction doit retourner **la valeur 0 sinon,** on appelle la fonction **nblivre** en passant en paramètre le **pointeur** sur la **liste des livres** écrits par **nom.**

```
fonction nblivraut (d bibli : biblio; d m : entier; d nom : ch20) : entier;
spécification {m ≥ 1,liste ordonnée des auteurs} =>{(nom ∈ bibli[0..m-1],
                        nblivraut = nombre de livres écrits par l'auteur nom)
                                  ∨ (nom ∉ bibli[0..m-1] , nblivraut = 0)}
var p : paut;
début
    p := pointaut (bibli, m, nom);
    si p = nil alors
        {auteur absent}
        nblivraut := 0
    sinon
        {auteur présent}
        nblivraut := nblivre(p↑.prem)
fin;
```

2. INSERTION D'ÉLÉMENTS

2.1. *INSERTION D'UN AUTEUR*

2.1.1. On s'inspire de la procédure **insertête** du cours (cf tome 2, p. 46), en n'oubliant pas d'initialiser les champs **prem** et **der** à **nil.**

```
procédure instêteaut (dr pa : paut ; d nom : ch20);
spécification {nom ∉ pa+} => {l'auteur nom qui n'a encore écrit aucun
                                   livre a été inséré en tête de pa+}
var p : paut;
```

```
début
    nouveau(p);
    p↑.nomaut := nom;
    p↑.prem := nil;
    p↑.der := nil;
    p↑.autsuiv := pa;
    pa := p
fin;
```

2.1.2. On s'inspire de la procédure **insertri** du cours (cf Tome2, p. 60).

procédure insaut (dr pa : paut ; d nom : ch20 ; r pauteur:paut);

spécification {**pa**$^+$ **triée**} => {(**nom** $\notin$ **pa**$^+$, l'auteur **nom** qui n'a encore écrit aucun livre a été inséré dans **pa**$^+$) $\vee$ (**nom** $\in$ **pa**$^+$, pas d'insertion) , **pauteur** = adresse de la cellule contenant **nom**}

```
début
    si pa = nil alors
    début
        {nom est absent => insertion}
        instêteaut (pa, nom);
        pauteur := pa
    fin
    sinon
        si pa↑.nomaut < nom alors
            insaut (pa↑.autsuiv, nom, pauteur)
        sinon
            si pa↑.nomaut > nom alors
            début
                {nom est absent => insertion}
                insfêteaut(pa, nom);
                pauteur := pa
            fin
            sinon
                {(pa↑.nomaut = nom) => nom est présent}
                pauteur := pa
fin;
```

2.1.3. Appel de insaut avec le pointeur sur la liste des livres écrits par nom.

procédure insauteur (dr bibli:biblio;d m:entier;d nom:ch20;r pauteur:paut);

spécification {**m** $\geq$ **1**} => {(**nom** $\notin$ **bibli[0..m-1]**, le nouvel auteur **nom** qui n'a encore écrit aucun livre a été inséré dans **bibli[0..m-1]**) $\vee$ (**nom** $\in$ **bibli[0..m-1]** , pas d'insertion) , **pauteur** = adresse de la cellule contenant **nom**}

```
début
    insaut (bibli [calculecode (nom, m) ], nom , pauteur)
fin;
```

2.2. *INSERTION D'UN LIVRE*

2.2.1.

Insertion en tête d'une liste (cf Tome2, p. 46).

```
procédure instêtelivre (dr pl : plivre ; d titre : ch20);
spécification { } => {le livre titre a été inséré en tête de pl+}
var  p : plivre;
début
      nouveau(p);
      p↑.titre := titre;
      p↑.titresuiv := pl;
      pl := p
fin;
```

2.2.2.

Insertion classique dans une file d'attente ; on s'inspire de la procédure **ajoutelem** défini dans le cours (cf Tome 2, p.120).

```
procédure inslivre (d pa : paut ; d titre : ch20);
spécification {pa ≠ nil} => {le dernier livre titre écrit par pa↑.nomaut a été
                              inséré dans la liste des livres écrits par pa↑.nomaut}
début
      si pa↑.der = nil alors
      {pa↑.prem = nil , l'auteur n'a encore écrit aucun livre}
      début
            instêtelivre(pa↑.der, titre);
            pa↑.prem := pa↑.der
            {le premier livre est aussi le dernier}
      fin
      sinon
      début
            instêtelivre(pa↑.der↑.titresuiv, titre);
            {insertion en fin de liste}
            pa↑.der := pa↑.der↑.titresuiv
            {mise à jour du pointeur sur le dernier}
      fin
fin;
```

2.2.3.

```
procédure inserlivre (dr bibli : biblio; d m : entier; d nom , titre : ch20);
spécification {m ≥ 1} => {le dernier titre écrit par nom ∈ bibli[0..m-1]}
var  pauteur : paut;
début
      insauteur(bibli, m, nom, pauteur);
      {l'auteur nom a été inséré si nécessaire , pauteur↑.nomaut = nom}
      inslivre(pauteur, titre)
      {le dernier livre titre écrit par nom a été inséré}
fin;
```

3. SUPPRESSIONS D'ÉLÉMENTS

3.1. *SUPPRESSION D'UN LIVRE*

3.1.1. Fonction classique définie dans le cours (cf Tome2, p. 51).

```
fonction dernier (d pl : plivre) : plivre;
spécification {pl ≠ nil} =>{dernier = adresse de la dernière cellule de pl+}
début
      si pl↑.titresuiv = nil alors
            dernier := pl
      sinon
            dernier := dernier (pl↑.titresuiv);
fin;
```

3.1.2.

```
procédure supptêtelivre (dr prem , der , pl : plivre) ;
spécification {pl+ non vide} => {la cellule de tête de pl+ a été supprimée ,
                                   prem et der ont été mis à jour si nécessaire}
var  p : plivre;
début
      p := pl;
      pl := pl↑.titresuiv;
      {en cas de suppression du premier, prem et pl désignent la même
                                                              variable}
      si pl = nil alors {suppression du dernier}
            si prem = nil alors {le dernier était aussi le premier}
                  der := nil
            sinon
            {ce n'était pas le premier, il faut donc recalculer l'adresse du
                                                              dernier}
                  der := dernier(prem);
      laisser(p);
fin;
```

3.1.3.

```
procédure suppunlivre (dr prem , der , pl : plivre ; d titre : ch20) ;
spécification {pl+ ordonnée} => {le livre titre de pl+ a été supprimé ,
                                   prem et der ont été mis à jour si nécessaire}
{ne fait rien si titre n'appartient pas à prem+}
début
      si pl ≠ nil alors
            si pl↑.titre = titre alors
                  supptêtelivre(prem, der, pl)
            sinon
                  suppunlivre(prem, der, pl↑.titresuiv , titre)
fin;
```

3.1.4.

procédure supplivreaut (d bibli : biblio; d m : entier ; d nom,titre : ch20);
spécification {**m** ≥ **1**} => {le livre **titre** écrit par **nom** ∉ **bibli[0..m-1]** ,
prem et **der** ont été mis à jour si nécessaire}
{n'a aucun effet si l'auteur n'a pas écrit ce livre ou si l'auteur n'existe pas}
var p : paut;
début
    p := pointaut(bibli, m, nom);
    si p ≠ nil alors *{p↑.nomaut = nom ,p↑.prem = adresse du premier livre, p↑.der = adresse du dernier livre}*
        suppunlivre (p↑.prem, p↑.der, p↑.prem, titre);
fin;

3.2. ***SUPPRESSION D'UN AUTEUR***

3.2.1. Suppression de tous les éléments d'une liste. Sous forme récursive, on utilise le deuxième parcours parcours2 (cf Tome 2, p. 17).

Version récursive :

procédure supplivres (dr pl : plivre);
spécification { } => {tous les livres de **pl+** ont été supprimés}
début
    si pl ≠ nil alors
    début
        supplivres (pl↑.titresuiv);
        laisser (pl);
    fin;
fin;

Version itérative :

procédure supplivres (dr pl : plivre);
spécification { } => {tous les livres de **pl+** ont été supprimés}
var p : plivre;
début
    tantque pl ≠ nil faire
    début
        p := pl ; *{p = adresse de la cellule courante}*
        pl := pl↑.titresuiv ; *{pl = adresse de la cellule suivante}*
        laisser(p);
        {la cellule d'adresse courante p a été rendue à la liste libre}
    fin;
fin;

3.2.2.

Suppression classique en tête de liste (cf Tome 2, p. 67).

procédure supptêteaut (dr pa : paut) ;
spécification {**pa ≠ nil**} => {la cellule de tête de **pa+** a été supprimée}
var p : paut;

```
début
    p := pa;
    pa := pa↑.autsuiv;
    laisser(p);
fin;
```

3.2.3.
Suppression, sous forme récursive, dans une liste triée.

```
procédure suppautl (dr pa : paut ; d nom : ch20) ;
spécification {pa+ triée} => {l'auteur nom et tous ses livres ont été
                                                    supprimés de pa+}
{ne fait rien si nom ∉ pa+}
début
    si pa ≠ nil alors
        si pa↑.nomaut = nom alors
        début
            {suppression de tous les livres}
            supplivres(pa↑.prem);
            {suppression de l'auteur}
            supptêteaut(pa);
        fin
        sinon
            si pa↑.nomaut < nom alors
                {appel récursif, pa+ est ordonnée}
                suppautl (pa↑.autsuiv, nom);
fin;
```

3.2.4.
On fait appel à la procédure précédente.

```
procédure suppauteur (dr bibli : biblio ; d m : entier; d nom : ch20) ;
spécification {m ≥ 1} => {l'auteur nom et tous ses livres ∉ bibli[0..m-1]}
{ne fait rien si nom ∉ bibli [0..m -1]}
début
    suppautl(bibli[calculecode(nom, m)], nom);
fin;
```

6.3. Inscriptions à l'université

(HASH-CODE RÉCURRENT)

Énoncé

On souhaite gérer la liste des étudiants inscrits en 1989 dans une UFR de l'Université de Grenoble. Pour cela, on dispose d'un fichier FETUD où figurent, dans l'ordre de leur numéro d'inscription, toutes les informations sur chaque étudiant.
On supposera, pour simplifier :
- que chaque enregistrement ne comporte que les informations suivantes : numéro d'inscription, nom et prénom, résultat final (admis ou ajourné) ; le résultat reste vide tant que l'examen n'a pas eu lieu ;
- qu'il n'y a jamais deux étudiants avec le même nom et le même prénom dans la même UFR.

Lorsque l'on veut consulter ou mettre à jour les renseignements concernant un étudiant, il peut être nécessaire d'accéder à l'enregistrement le concernant en donnant son nom plutôt que son numéro d'inscription. Pour cela, on construit une structure permettant l'accès direct au fichier sur les noms. Cette structure devra être tenue à jour lors de l'inscription d'un nouvel étudiant.

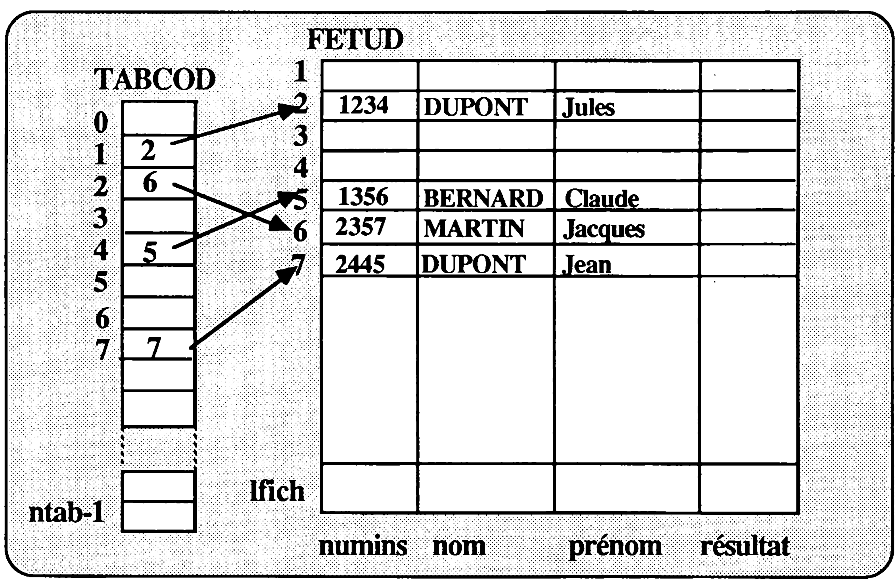


Figure 1

Il s'agit (cf Fig.1) d'une table **tabcod** qui est gérée en adressage dispersé, avec résolution récurrente des collisions ("hash-code récurrent" cf Tome 2, p. 182). On accède directement à cette table par une clé calculée au moyen du nom de l'étudiant. Elle contient, pour chaque étudiant, son numéro d'enregistrement dans **fétud**. Le choix de cette organisation se justifie par les accès rapides qu'elle autorise, et par le fait que les suppressions dans le fichier seront rares ou inexistantes.

Dans l'exemple de la Fig.1, avec un pas de récurrence égal à 3, DUPONT et BERNARD ont la même clé égale à 1, alors que MARTIN a la clé 2. Pour simplifier l'écriture des algorithmes, on simulera le fichier **fétud** par un tableau, avec accès direct indicé, et rangement des nouveaux étudiants à la fin, après le dernier indice utilisé.

Les déclarations peuvent alors s'écrire ainsi :

```
const  ntab    = ... ;         {nombre d'éléments de tabcod}
       lfich   = ... ;         {longueur de fétud}
       pas     = ... ;         {pas de récurrence}

type   code    =  0..ntab-1;   {indices dans tabcod}
       adr     =  0..lfich;    {indices dans fétud}
       lnom    =  0..20;
       étud    =  structure
                     numins : chaîne4;
                     nom, prénom : chaîne20;
                     résultat : chaîne4;
                  fin;
       tétud   =  tableau [1..lfich] de étud;
       tcod    =  tableau [code] de adr;

var    fétud   :  tétud;
       tabcod  :  tcod;
       dernier :  adr;         {dernier indice utilisé dans fétud}
```

Remarques :

- on considèrera dans la suite que les variables déclarées ci-dessus sont globales, et pourront donc ne pas figurer parmi les paramètres des algorithmes qui les traitent ;

- on supposera dans les algorithmes demandés que la table **tabcod** a été initialisée avec des 0 dans tous ses éléments, et que toutes les zones de **fétud** ont été initialisées avec des chaînes vides.

1. On pense qu'il y aura au plus 2000 étudiants dans l'UFR. Quelles valeurs peut-on envisager de donner à **ntab**, à **pas**, et à **lfich** ?

2. Écrire une
fonction clé (d nom : chaîne20) : code;
qui calcule la clé associée à un identificateur **nom** en prenant la somme modulo **ntab** des codes ASCII des caractères de **nom.**

3. Écrire une
procédure affich (d nom : chaîne20);
qui affiche toutes les occurrences de **nom**, avec pour chacune le numéro d'inscription, le prénom et le résultat aux examens, s'il y a lieu. Si aucun étudiant **nom** n'est inscrit, la procédure affichera un message.

Exemple : affich ('DUPONT') permettrait d'afficher :

```
1234  DUPONT Jules
2445  DUPONT Jean
```

On notera que toutes les occurrences de nom ont évidemment la même clé, mais que d'autres noms peuvent avoir cette clé ("collision").

4. Écrire une
fonction rangfich (d numéro : entier; d nom, prénom : chaîne20) : booléen;
qui vérifie si **numéro** est bien supérieur au numéro d'inscription du dernier étudiant inscrit dans **fétud.**
Si c'est vrai, la fonction enregistre le numéro, ainsi que **nom** et **prénom**, dans **fétud**, à la suite des enregistrements précédents ; la variable globale **dernier** doit alors être mise à jour. On supposera, pour simplifier, que, par analogie avec un fichier, il reste toujours de la place dans **fétud.**

5. Écrire une
fonction recherche (d nom, prénom : chaîne20; r place : code) : booléen;
qui recherche l'étudiant **nom prénom** dans **fétud**, dans la classe d'équivalence de **nom** ; si cet étudiant est inscrit, recherche délivre **vrai**, et range dans **place** l'indice correspondant de **tabcod**; si l'étudiant n'est pas inscrit, recherche délivre **faux**, et range dans **place** l'indice du premier emplacement libre dans **tabcod** pour la classe d'équivalence de **nom**; si la table est pleine, **place** sera égal à 0.

6. Écrire une
procédure majrésul (d nom, prénom : chaîne20; d résultat : chaîne4);
qui met à jour l'enregistrement relatif à l'étudiant **nom prénom**, en y mettant le **résultat** obtenu par cet étudiant.
La procédure devra afficher un message indiquant que la mise à jour a été effectuée, ou bien qu'elle n'a pu être effectuée en raison d'une erreur sur l'identité de l'étudiant.

7. Écrire une
procédure inscrire (d nom, prénom : chaîne20; d numéro : entier);
qui range dans **fétud** les données relatives à nouvel étudiant, si cette inscription est possible.
Elle devra de plus afficher un message signalant que l'inscription a été effectuée, ou bien un message différent pour chaque cas d'erreur.

Solutions proposées

1. Rappelons que le taux de remplissage d'une table gérée en "hash-code récurrent" ne doit pas dépasser 70% environ, si l'on veut que les temps d'accès demeurent courts. On doit donc donner à **ntab** une valeur supérieure à 2000/0.7, soit **ntab > 2857**. Quant à la valeur du **pas**, il suffit de prendre un entier premier avec **ntab**. On peut donc proposer :
ntab = 2900 et pas = 3, ou bien **ntab = 3000** et **pas** = 7...

La constante **lfich** ne sert que dans la simulation d'un fichier par un tableau, il suffit qu'elle soit égale au nombre maximum d'étudiants, soit
lfich = 2000.

2. On supposera qu'il existe une primitive "longueur" qui délivre le nombre de caractères d'une variable de type chaîne.

```
fonction clé (d nom : chaîne20) : code;
var  i, c : entier;
début
     c := 0;
     pour i := 1 à longueur(nom) faire
          c := c + ord (nom [i]);
     clé := c mod ntab;
fin;
```

3. Dans cette procédure, on utilisera un booléen **fini** qui permet d'arrêter la recherche dans deux cas :
- lorsqu'on rencontre un élément nul dans **tabcod**, c'est que l'on a terminé le parcours des éléments ayant la même clé (classe d'équivalence);
- lorsque le code est égal à la clé calculée initialement, c'est que l'on a parcouru tous les éléments de **tabcod** ; ce cas ne devrait jamais se présenter si la taille de **tabcod** est suffisante.

```
procédure affich (d nom : chaîne20);
var  cod, codinit : code;
     fini : booléen;
     étudiant : étud;
     n : entier;
```

```
début
      cod := clé (nom);
      codinit := cod;
      fini := faux;
      n := 0;
      tantque non fini faire
      début
            si tabcod[cod] = 0 alors
                  fini := vrai
            sinon
            début
                  étudiant := fétud[tabcod[cod]];
                  si étudiant .nom = nom alors
                  début
                        avec étudiant faire
                        écrireln (numins, nom,  prénom,  résultat);
                        n := n + 1
                  fin;
                  cod := (cod + pas) mod ntab;
                  si cod = codinit alors
                        fini := vrai
            fin
      fin;
      si n = 0 alors
            écrireln('pas de ', nom, ' inscrit')
fin;
```

4.

```
fonction rangfich (d numéro : entier; d nom, prénom : chaîne20) : booléen;
var possible: booléen;
début
      si dernier = 0 alors
            possible := vrai
      sinon
            possible := numéro > fétud [dernier].numins;
      si possible alors
      début
            dernier := dernier + 1;
            fétud [dernier].numins := numéro;
            fétud [dernier].nom := nom;
            fétud [dernier].prénom := prénom
      fin;
      rangfich := possible
fin;
```

5.

Il faudra, comme dans la fonction **affich**, faire arrêter la recherche lorsque la classe d'équivalence est entièrement parcourue, mais il faudra ici différencier le résultat dans **place** selon que la table est pleine ou non ; de plus, il faudra arrêter dès que l'on aura trouvé l'enregistrement cherché. On pourra donc employer trois booléens correspondant à ces trois cas.

```
fonction recherche (d nom, prénom : chaîne20; r place : code) : booléen;
var   cod, codinit: code;
      fini, plein, trouvé: booléen;
début
      cod := clé (nom);
      codinit := cod;
      fini := faux;
      plein := faux;
      trouvé := faux;
      tantque non fini et non plein et non trouvé faire
      début
            si tabcod[cod] = 0 alors
                  fini := vrai
            sinon
                  si (fétud [tabcod [cod] ] . nom = nom) et
                                    (fétud[tabcod[cod]].prénom = prénom) alors
                        trouvé := vrai
                  sinon
                  début
                        cod := (cod + pas) mod ntab;
                        si cod = codinit alors  plein := vrai
                  fin;
      fin;
      recherche := trouvé;
      place := cod;
      si plein alors place := 0
fin;
```

6.

```
procédure majresul (d nom, prénom : chaîne20; d résultat : chaîne4);
var place : code;
début
      si recherche (nom, prénom, place) alors
      début
            fétud [tabcod [place] ] . résultat := résultat;
            écrireln (' résultat enregistré')
      fin
      sinon
            écrireln (' erreur, étudiant non inscrit')
fin;
```

7.

```
procédure inscrire (d nom, prénom : chaîne20;  d numéro : entier);
var place : code;
début
      si recherche (nom, prénom, place) alors
            écrireln ('erreur, étudiant déjà inscrit ')
      sinon
            si place = 0 alors
                  écrireln ('plus de place pour insérer')
            sinon
                  si rangfich (numéro, nom, prénom) alors
                  début
                        tabcod [place] := dernier;
                        écrireln ('inscription effectuée')
                  fin
                  sinon
                        écrireln ('erreur sur le numéro d''inscription')
fin;
```

6.4. Classements d'un concours

(SIMULATION DE CHAINAGES)

Énoncé

On se propose de gérer la saisie et l'affichage des différents classements d'un concours de gymnastique. Chaque concurrent sera identifié par son nom, supposé unique, et aura une note globale indiquant ses résultats au concours. On connaît aussi l'âge, en années, de chaque concurrent. L'on souhaite disposer des tableaux suivants :

- liste classée par ordre alphabétique, avec indication de l'âge et de la note obtenue par chaque concurrent ;
- liste classée par ordre croissant d'âge des concurrents, et par ordre alphabétique à âge égal, avec indication de leur rang dans le classement ;
- liste classée par ordre décroissant des notes (ordre croissant des rangs).

Exemple :
Supposons qu'il y ait 10 concurrents, et qu'on ait saisi dans un ordre quelconque leur nom, leur âge et leur note.

On affichera les tableaux suivants :

Ordre alphabétique :

Nom	*Age*	*Note*
Abel	14	9.2
Adam	18	6.3
Babel	15	9.6
Caïn	18	6.3
David	13	7.4
Eve	16	5.6
Goliath	17	9.5
Jéricho	14	4.5
Josué	16	5.0
Moïse	17	8.7

Classement par âge :

Age	*Nom*	*Rang*
13	David	5
14	Abel	3
14	Jéricho	10
15	Babel	1
16	Eve	8
16	Josué	9
17	Goliath	2
17	Moïse	4
18	Adam	6
18	Caïn	6

Classement par notes :

Rang	*Nom*	*Note*
1	Babel	9.6
2	Goliath	9.5
3	Abel	9.2
4	Moïse	8.7
5	David	7.4
6	Adam	6.3
6	Caïn	6.3
8	Eve	5.6
9	Josué	5.0
10	Jéricho	4.5

Pour ne pas avoir à stocker plusieurs fois les noms des concurrents, la structure de données interne sera la suivante :

Dans un vecteur d'enregistrements **concours**, on rangera, au fur et à mesure de la saisie, les données de chaque concurrent. Ainsi, **nom**, **âge** et **note** occuperont chacun un champ de l'enregistrement.

Les classements seront créés au moyen de plusieurs champs supplémentaires, qui contiendront les indices qui permettent de simuler la création de listes chaînées, triées selon les critères choisis. Il y a trois types de classement : à chacun correspondra un champ (**suiv1** pour l'ordre alphabétique, **suiv2** pour l'ordre des âges, **suiv3** pour le classement sur les notes). A la saisie de chaque concurrent, on effectuera une mise à jour des pointeurs, en insérant ce concurrent parmi les autres, déjà classés. On créera également un champ **rang** permettant de connaître le rang d'un concurrent parmi ceux de sa catégorie.

Chacune des listes sera déterminée par l'indice de son élément de tête, rangé dans une variable entière : **liste1, liste2** et **liste3.**

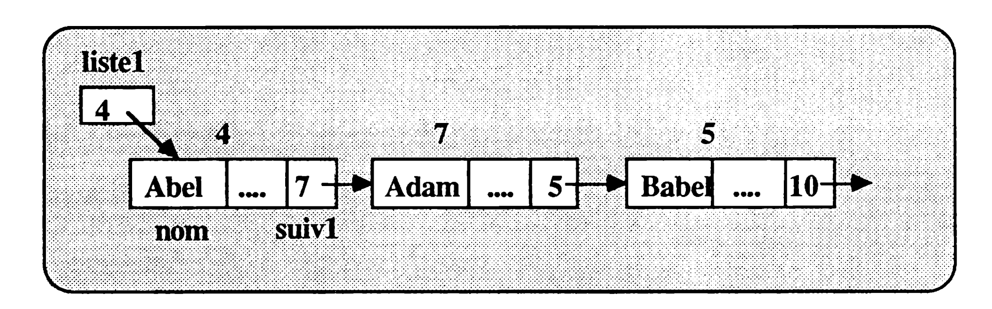
Le tableau final que l'on obtiendrait avec les données ci-dessus serait le suivant (l'ordre des noms est celui, tout à fait aléatoire, de la saisie) :

CONCOURS :

	nom	âge	note	rang	suiv1	suiv2	suiv3
1	Moïse	17	8.7	4	0	7	3
2	Josué	16	5.0	9	1	9	6
3	David	13	7.4	5	8	4	7
4	Abel	14	9.2	3	7	6	1
5	Babel	15	9.6	1	10	8	9
6	Jéricho	14	4.5	10	2	5	0
7	Adam	18	6.3	6	5	10	10
8	Eve	16	5.6	8	9	2	2
9	Goliath	17	9.5	2	6	1	4
10	Caïn	18	6.3	6	3	0	8

Dans ce tableau, nil est représenté par 0. Les valeurs des pointeurs de tête de liste sont : **liste1= 4, liste2 = 3** et **liste3= 5.**

Ainsi par exemple, à l'aide de **liste1** et des pointeurs **suiv1**, on a simulé la liste suivante, qui correspond au classement en ordre alphabétique :



On utilisera les déclarations de type suivantes :

type **ch10 = chaîne10 ;**
concurrent = structure
nom : ch10;
âge : entier ;
note : réel;
rang :entier ;*{rang : dans le classement par notes}*
suiv1, suiv2, suiv3 : entier;
{suiv1 : pointeur ordre alphabétique ;
suiv2 : pointeur ordre par âge;
suiv3 : pointeur classement par notes}
fin;
tconcours = tableau [1..100] de concurrent;
list1, list2, list3 : entier;

Écrire les algorithmes suivants :

1.

procédure inser (d nom : ch10; d note : réel ; d âge : entier; dr concours : tconcours; dr nb : entier);
spécification {**nb** ≥ 0} => {un nouvel enregistrement contenant **nom**, **note** et **âge** a été inséré à la fin de **concours [1 .. nb]**}

2. On veut maintenant simuler l'insertion d'un nouvel enregistrement dans une liste chaînée triée. Pour cela, on devra modifier certains indices des zones **suiv1, suiv2** ou **suiv3**, selon le critère de tri choisi. L'indice de la tête de liste sera chaque fois un paramètre modifiable de la procédure correspondante. Toutes ces procédures seront récursives.

2.1.

procédure maj1 (dr concours : tconcours ; d nb : entier; dr list : entier);
spécification {**nb** ≥ 0 ; **list** = indice de la tête de liste du classement alphabétique par noms} => {un nouvel enregistrement **concours [nb]** a été inséré, dans l'ordre alphabétique des noms, parmi les enregistrements de **concours [1 ..nb - 1]**}

2.2.
procédure maj2 (dr concours : tconcours; d nb : entier; dr list : entier);
spécification {**nb** $\geq$ 0 ; **list** = indice de la tête de liste du classement par âges} => {un nouvel enregistrement **concours [nb]**, a été inséré dans l'ordre des âges, et à âge égal par ordre alphabétique des noms, parmi les enregistrements de **concours [1 ..nb - 1]**}

2.3.
procédure maj3 (dr concours : tconcours; d nb : entier; dr list : entier);
spécification
{**nb** $\geq$ 0 ; **list** = indice de la tête de liste du classement en ordre décroissant des notes} => {un nouvel enregistrement **concours [nb]** a été inséré, dans le même ordre, parmi les enregistrements de **concours [1 ..nb - 1]**}

3.
procédure calculrang (dr concours : tconcours; d list : entier);
spécification {**list** = indice de la tête de liste du classement par notes} => {calcul des rangs dans ce classement}

Cette procédure calcule le rang de chaque concurrent dans le classement par notes, en tenant compte des ex aequo. Pour cela, on compare les notes des concurrents successifs à l'aide des pointeurs **suiv3**.

4. Les procédures suivantes permettront, une fois tous les concurrents saisis et tous les pointeurs calculés, d'afficher les classements souhaités.

4.1.
procédure affich1(d concours:tconcours; d list: entier);
spécification {**list** = tête de liste du classement alphabétique} => {affichage de ce classement}

4.2.
procédure affich2(d concours:tconcours; d list : entier);
spécification {**list** = tête de liste du classement par âges} => {affichage de ce classement}

4.3.
procédure affich3 (d concours:tconcours; d list : entier);
spécification {**list** = tête de liste du classement par notes} => {affichage de ce classement}

5. On souhaite enfin enchaîner toutes les opérations décrites ci-dessus, afin d'obtenir la saisie des données concernant tous les concurrents, le calcul et l'affichage des classements, et enfin la sauvegarde du tableau contenant tous ces résultats et des têtes de listes. Pour cela, on écrira la procédure suivante :
procédure résultats_concours (r concours : tconcours; r nb : entier ; r liste1, liste2, liste3 : entier);
On supposera qu'il existe des procédures **saisienom** et **saisienoteage** qui permettent de saisir un nom, une note et un âge, et que la fin de la saisie est signalée par un nom "vide".

Solutions proposées

1. Il suffit d'insérer à la fin du tableau, en mettant à jour **nb**.

```
procédure inser (d nom : ch10; d note : réel ; d âge : entier;
                                        dr concours : tconcours; dr nb : entier);
spécification {nb ≥ 0} => {un nouvel enregistrement contenant nom, note
                                  et âge a été inséré à la fin de concours [ 1 .. nb ]}
début
    nb := nb+1;
    concours [nb] . nom := nom;
    concours [nb] . note := note;
    concours [nb] . âge := âge;
fin;
```

2.1. On reprend ici l'algorithme classique d'insertion d'un élément dans une liste chaînée triée (cf Tome 2, p. 60), en remplaçant les pointeurs sur l'élément suivant par la zone **suiv1** des enregistrements (cf Tome 2, p. 140).

```
procédure maj1 (dr concours : tconcours ; d nb : entier; dr list : entier);
spécification {nb ≥ 0 ; list = indice de la tête de liste du classement
alphabétique par noms} => {un nouvel enregistrement concours [ nb ] a été
    inséré, dans l'ordre alphabétique des noms, parmi les enregistrements de
                                                        concours [ 1 ..nb - 1 ]}
début
    si list = 0 alors
    début {insertion en fin de liste}
        concours [nb].suiv1:= list;
        list := nb
    fin
    sinon
        si concours [list].nom > concours [nb].nom alors
        début {insertion en tête des noms plus grands}
            concours [nb].suiv1 := list;
            list := nb
        fin
        sinon
            maj1(concours, nb, concours [list].suiv1)
fin;
```

2.2. L'algorithme est semblable à celui de la question précédente, en remplaçant **suiv1** par **suiv2**, et en modifiant le critère de tri.

```
procédure maj2 (dr concours : tconcours; d nb : entier; dr list : entier);
spécification {nb ≥ 0 ; list = indice de la tête de liste du classement par âges}
  => {un nouvel enregistrement concours [ nb ] , a été inséré dans l'ordre des
              âges, et à âge égal par ordre alphabétique des noms, parmi les
                                     enregistrements de concours [ 1 ..nb - 1 ]}
```

```
début
    si list = 0 alors
    début
        concours [nb].suiv2 := list;
        list := nb
    fin
    sinon
        si (concours [list].âge > concours [nb].âge)  ou
            ((concours [list].âge = concours [nb].âge)  et
                (concours [list].nom > concours [nb].nom)) alors
        début
            concours [nb].suiv2 := list;
            list := nb
        fin
        sinon  maj2 (concours, nb, concours [list].suiv2)
fin;
```

2.3. L'algorithme est encore semblable à celui de la question **2.1.**, en remplaçant **suiv1** par **suiv3**, et le test sur les noms croissants par un test sur les notes décroissantes.

```
procédure maj3 (dr concours : tconcours; d nb : entier; dr list : entier);
spécification
{nb ≥ 0 ; list = indice de la tête de liste du classement en ordre décroissant
des notes}  =>  {un nouvel enregistrement concours [nb]  a été inséré, dans
        le même ordre, parmi les enregistrements de concours [1 ..nb - 1 ]}
début
    si list = 0 alors
    début
        concours [nb].suiv3 := list;
        list := nb
    fin
    sinon
        si concours [list].note < concours [nb].note alors
        début
            concours [nb].suiv3 := list;
            list := nb;
        fin
        sinon  maj3(concours, nb, concours [list] . suiv3)
fin;
```

3.

```
procédure calculrang (dr concours : tconcours; d  list : entier);
spécification {list =  indice de la tête de liste du classement par notes} =>
                                        {calcul des rangs dans ce classement}
var   rang, n : entier;
    note : réel ;
```

```
début
    rang := 1; {initialisation du rang courant}
    n := 1; {initialisation du  numéro d'ordre dans le classement}
    note := concours [list].note; {initialisation de la note courante}
    tantque list ≠ 0 faire
    début
        si concours [list] .note < note alors
        début  {mise à jour du rang et de la note courante}
            rang := n;
            note := concours [list] .note
        fin;
        concours [list].rang := rang;
                              {insertion du rang courant dans le tableau}
        list := concours [list].suiv3;
        n := n+1 {mise à jour du  numéro d'ordre}
    fin;
fin;
```

4.1. Il s'agit du parcours classique d'une liste chaînée, où l'on a remplacé les pointeurs par la zone **suiv1** des enregistrements.

```
procédure affich1 (d concours:tconcours; d  list: entier);
spécification {list = tête de liste du classement alphabétique}
                                        => {affichage de ce classement}
début
    écrireln ('Ordre alphabétique :');  écrireln('Nom       Age     Note');
    tantque list ≠ 0 faire
    début
        écrireln (concours [list].nom , concours [list] . âge ,
                                                concours [list] . note);
        list := concours [list].suiv1
    fin;
fin;
```

4.2. C'est la même principe que ci-dessus, en remplaçant **suiv1** par **suiv2.**

```
procédure affich2 (d concours:tconcours; d  list : entier);
spécification {list = tête de liste du classement par âges}
                                        => {affichage de ce classement}
début
    écrireln ('Classement par âge:');  écrireln('Age   Nom     Rang ');
    tantque list ≠ 0 faire
    début
        écrireln (concours [list].âge, concours [list]. nom,
                                                concours [list].rang);
        list := concours [list].suiv2
    fin;
fin;
```

4.3. C'est encore le même algorithme que ci-dessus, en remplaçant **suiv2** par **suiv3.**

```
procédure affich3 (d concours:tconcours; d  list : entier);
spécification {list = tête de liste du classement par notes}
                                        => {affichage de ce classement}
début
    écrireln ('Classement par notes:'); écrireln ('Rang  Nom   Note ');
    tantque list ≠ 0 faire
    début
        écrireln (concours [list].rang, concours [list]. nom,
                                                  concours [list].note);
        list := concours [list].suiv3
    fin;
fin;
```

5. Cette procédure comporte trois phases : initialisation de **nb** et des têtes de listes, puis itération sur la saisie et le calcul des classements, et enfin calcul des rangs et affichage des résultats.

```
procédure résultats_concours (r  concours : tconcours; r nb : entier ;
                                          r liste1, liste2, liste3 : entier);
var  nom : ch10;
     note : réel;
     âge : entier;
début
    {initialisations}
    nb := 0;
    liste1 := 0; liste2 := 0; liste3:= 0;
    {itération}
    saisienom (nom) ;
    tantque nom ≠ '' faire
    début
        saisienoteage (note, âge);
        inser(nom, note, âge , concours, nb);
        maj1(concours, nb, liste1);
        maj2(concours, nb, liste2);
        maj3(concours, nb, liste3);
        saisienom (nom)
    fin;
    calculrang (concours , liste3);
    {affichages}
    affich1(concours,  liste1);
    affich2(concours,  liste2);
    affich3(concours,  liste3)
fin;
```

6.5. Réseau du métro

(VECTEUR DE LISTES)

Énoncé

Soit la structure suivante représentant un réseau du métro :

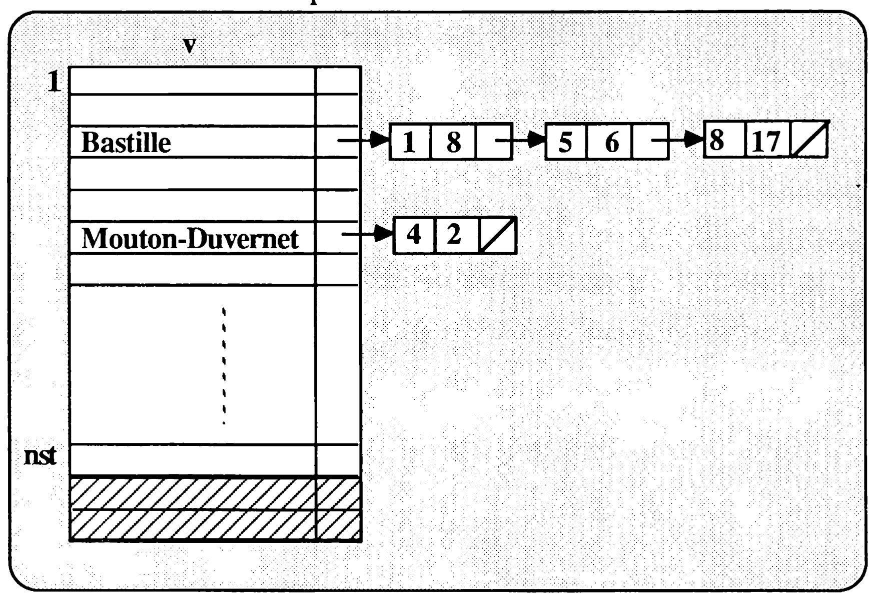


Figure 1

Le vecteur **v** est de type **vstat**, avec les déclarations suivantes :

```
type    nomstat  =   chaîne20;
        tstat    =   structure
                          station : nomstat;
                          adr : pointeur
                       fin;
        pointeur=    ↑cellule;
        cellule  =   structure
                          numligne, rang : entier;
                          suivant : pointeur
                     fin;
        vstat    =   tableau [ 1 .. 100 ] de tstat;
```

Le vecteur **v** est ordonné par ordre alphabétique sur les noms des stations de type nomstat (20 caractères au plus). Le nombre de stations présentes dans **v** est indiqué par l'entier **nst** .

A chaque station, on fait correspondre un pointeur sur une liste indiquant, pour chaque ligne passant par la station, son numéro d'ordre dans la ligne. Cette liste est ordonnée sur les numéros de lignes.
Exemple : la station Bastille se trouve sur trois lignes :
elle est la 8ème station sur la ligne no 1
elle est la 6ème station sur la ligne no 5
elle est la 17ème station sur la ligne no 8.
De même, Mouton-Duvernet est la 2ème station sur la ligne no 4.

1. CRÉATION DE LA STRUCTURE À PARTIR D'UN FICHIER SÉQUENTIEL

1.1. Écrire une
procédure insert (dr liste : pointeur; d noligne, ordre : entier);
spécification {**liste**+ triée} => {une cellule comportant la ligne de numéro **noligne** et de rang **ordre** a été insérée dans **liste**+}

a) Algorithme itératif
b) Algorithme récursif

1.2. En utilisant un parcours **séquentiel**, écrire une
fonction appart (d v : vstat; d nst : entier; d stat : nomstat; r place : entier) : booléen;
spécification { } =>
{(¬ **appart, stat** ∉ **v[1..nst],place** = rang auquel il faudra insérer **stat** dans v) ∨ (**appart, stat = v [place].station**)}

1.3. Écrire la fonction précédente en utilisant un parcours **dichotomique**.
a) Raisonnement par récurrence
b) Algorithme itératif
c) Algorithme récursif.

1.4. Écrire une
procédure insertion (dr v: vstat; dr nst: entier; d stat: nomstat; d noligne, ordre: entier);
spécification { } => {la station **stat**, qui se trouve à la place **ordre** sur la ligne **noligne,** a été insérée dans **v[1..nst]**}

1.5. Soit un fichier séquentiel f de type **fstat**, dont les enregistrements contiennent un nom de station, un numéro de ligne, et le rang de la station sur cette ligne : **type fstat = fichier de structure**
station : nomstat;
numligne, rang : entier;
fin;

Le fichier **f**, non trié, contient tous les noms de stations du métro ; lorsqu'une station appartient à plusieurs lignes, le fichier **f** contiendra plusieurs enregistrements ayant le même nom de station. Ces enregistrements ne sont ni forcément consécutifs, ni triés sur **numligne**.
On demande d'écrire une
procédure consv (dr f : fstat ; dr v : vstat; dr nst : entier) ;
qui construit la structure représentant le réseau complet à partir du fichier séquentiel f.

2. EXPLOITATION DE LA STRUCTURE CRÉÉE DANS LA PREMIERE PARTIE

2.1. On rappelle qu'une station est dite "de correspondance" si elle appartient à au moins deux lignes. Écrire une
fonction corresp (d v : vstat; d nst : entier; d stat : nomstat) : booléen;
spécification { } => {**corresp = stat** est une station de correspondance}

2.2. Écrire une
fonction nbcorr (d v : vstat; d nst : entier) : entier;
spécification { } => {**nbcorr** = nombre de stations de correspondance}
a) Algorithme itératif
b) Algorithme récursif

2.3. Écrire une
fonction appartl (d liste : pointeur; d noligne : entier) : booléen;
spécification { } => {**appartl = noligne** $\in$ **liste+**}

2.4. Écrire une
fonction nbstat (d v : vstat ; d nst, noligne : entier) : entier;
spécification { } => {**nbstat** = nombre de stations sur la ligne **noligne**}

2.5. Écrire une
fonction présent (d v : vstat ; d nst : entier; d stat: nomstat; d noligne: entier) : booléen;
spécification { } => {**présent = stat** appartient à la ligne **noligne**}

2.6. Écrire une
fonction mêmeligne (d v : vstat ; d nst : entier; d stat1, stat2 : nomstat) : booléen;
spécification { } =>
{**mêmeligne = stat1** et **stat2** appartiennent à une même ligne}

2.7. On suppose que les lignes sont numérotées de 1 à N, sans "trous". Écrire une
fonction nblignes (d v: vstat; d nst: entier) : entier;
spécification { } => {**nblignes** = nombre de lignes du réseau}

3. CRÉATION D'UNE NOUVELLE STRUCTURE

3.1. On veut construire une liste chaînée représentant une ligne de métro.

Chaque élément de la liste est composé de :
- un nom de station,
- un indicateur précisant si la station permet les correspondances,
- un pointeur sur l'élément suivant.

On utilisera de nouveaux types :

type pointeur2 = ↑cellule2;
cellule2 = structure
station : nomstat;
cor : booléen;
suivant : pointeur2
fin;

Les stations figurent dans cette liste dans l'ordre de leur rang :

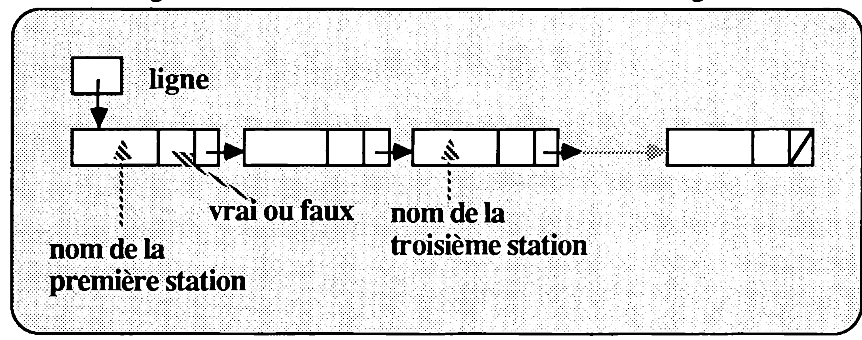


Figure 2

Écrire une
procédure consl (d v : vstat ; d nst, noligne : entier; r ligne : pointeur2);
qui construit une liste d'adresse **ligne** représentant la ligne de numéro **noligne**.

3.2. On veut, à partir du vecteur v, construire une nouvelle représentation du réseau (cf Fig. 3):

Écrire une
procédure conslignes (d v : vstat; d nst : entier; r lignes : vectligne; r n : entier);
avec
type vectligne = tableau [1..100] de pointeur2;

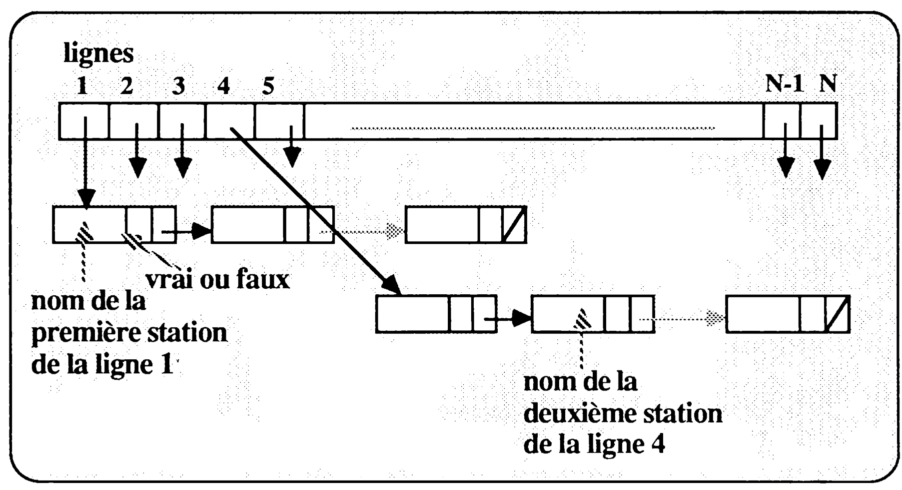


Figure 3

Solutions proposées

1. CRÉATION DE LA STRUCTURE À PARTIR D'UN FICHIER SÉQUENTIEL

1.1. a) Nous commençons par définir une procédure d'insertion en tête de liste (Tome 2, p. 46) qui nous servira dans les deux versions de la procédure d'insertion dans une liste triée :

procédure instête (dr liste : pointeur; d noligne, ordre : entier);
spécification { } => {une cellule comportant la ligne de numéro **noligne** et de rang **ordre** a été insérée en tête de **liste+**}
var p: pointeur;
début
nouveau(p);
p↑. numligne := noligne;
p↑.rang := ordre;
p↑. suivant := liste;
liste := p
fin;

Il ne reste alors plus qu'à parcourir la liste en cherchant la cellule d'adresse **préced**, qui précède la place d'insertion. Il faut prévoir les cas particuliers de l'insertion en tête de liste, et de la liste initiale vide : ce sont les cas où la variable auxiliaire **liste1** a gardé la valeur **liste**.

```
procédure insert (dr liste : pointeur; d noligne, ordre : entier);
spécification {liste+ triée} => {une cellule comportant la ligne de numéro
                        noligne et de rang ordre a été insérée dans liste+}
var  liste1, préced : pointeur;
     infer: booléen ;
début
     infer := vrai;
     liste1 := liste;
     tantque (liste1 ≠ nil) et infer faire
          si liste1↑.numligne < noligne alors
          début
               préced := liste1;
               liste1 := liste1↑.suivant
          fin
          sinon  infer := faux;
     {liste1 = nil ou  non infer}
     si liste1 = liste {insertion en tête de liste} alors
          instête (liste, noligne, ordre)
     sinon  {insertion après préced}
          instête (préced↑.suivant , noligne, ordre)
fin;
```

1.1. b) C'est l'algorithme classique (Tome 2, p. 60) :

```
procédure insert (dr liste : pointeur; d noligne, ordre : entier);
spécification {liste+ triée} => {une cellule comportant la ligne de numéro
                        noligne et de rang ordre a été insérée dans liste+}
début
     si liste = nil alors
          instête (liste, noligne, ordre)
     sinon
          si noligne ≤ liste↑.numligne alors
               instête (liste, noligne, ordre)
          sinon
               insert (liste↑.suivant, noligne, ordre)
fin;
```

1.2. Nous proposons ici la solution sans booléen, où l'on arrête l'itération, soit sur le premier élément supérieur ou égal à celui que l'on cherche, soit sur le dernier élément. Il faut penser à traiter séparément le cas où le vecteur est initialement vide.

```
fonction appart (d v : vstat; d nst : entier; d stat : nomstat; r place : entier)
                                                            : booléen;
spécification { } =>
{(¬ appart, stat ∉ v[1..nst],place = rang auquel il faudra insérer stat  dans v)
                                   ∨ (appart, stat = v [place].station)}
```

```
var   i: entier;
début
      appart := faux;
      si nst = 0 alors  {vecteur vide}
            place :=1
      sinon {nst > 0}
            si stat > v [ nst] . station alors
                  place := nst +1
            sinon {stat ≤ v[ nst] . station}
            début
                  i := 1;
                  tantque  stat >v [i].station  faire i := i + 1;
                  {stat ≤ v [i].station}
                  appart := v [i].station = stat;
                  place := i
            fin
fin;
```

1.3. a) Notons d'abord que la post-condition peut s'écrire :

v [place - 1] . station < stat ≤ v [place] . station

Soient alors deux valeurs **inf** et **sup** qui vérifient l'assertion A suivante :

A : {v [inf-1] .station < stat ≤ v [sup] . station}

Deux cas sont possibles :

. **inf = sup** alors la post-condition est vérifiée pour **place = inf**

. **inf < sup**

on pose alors **mil := (inf + sup) div 2**

.. **stat ≤ v [mil] . station**

alors **sup := mil** permet de retrouver l'assertion A

.. **stat > v [mil] . station**

alors **inf := mil+1** permet de retrouver l'assertion A

De ce raisonnement, on déduit soit l'itération, soit la procédure récursive.

Initialisation :

On élimine d'abord, comme ci-dessus, les cas où **nst = 0** ou bien **stat >v [nst] . station** , il suffit ensuite de poser **inf := 1** et **sup := nst .**

1.3. b)

```
fonction appart (d v: vstat; d nst: entier; d stat: nomstat;
                                        r place: entier) : booléen;
spécification { } =>
{(¬ appart, stat ∉ v[1..nst],place = rang auquel il faudra insérer stat  dans v)
                                  ∨ (appart, stat = v [place].station)}
```

```
var inf, sup, mil : entier;
début
    appart := faux;
    si nst = 0 alors {vecteur vide}
        place := 1
    sinon
        si stat > v[nst] . station  alors place := nst + 1
        sinon
        début
            inf := 1;  sup := nst;
            {v [inf-1] . station< stat ≤ v [sup] . station}
            tantque inf < sup faire
            début
                mil := (inf + sup) div 2;
                si stat ≤ v [mil] . station  alors
                    sup := mil
                sinon
                    inf := mil + 1
            fin;
            {inf = sup, v [inf-1].station < stat ≤v [sup] . station}
            place := inf;
            appart := stat = v [inf] .station ;
        fin;
fin;
```

1.3. c) Pour la version récursive, il faut éviter d'effectuer les tests initiaux à chaque itération. On est donc conduit à écrire une fonction principale qui fait les initialisations et appelle une fonction secondaire récursive.

```
fonction appart (d v: vstat;  d nst : entier; d stat : nomstat;
                                        r place : entier): booléen;
spécification { } =>
{(¬ appart, stat ∉ v[1..nst],place = rang auquel il faudra insérer stat  dans v)
                                   ∨ (appart, stat = v [place].station)}
début
    appart := faux;
    si nst = 0 alors {vecteur vide}
        place := 1
    sinon
        si stat > v [nst] . station  alors
            place := nst + 1
        sinon
            appart := appartrec (v,1, nst, stat, place)
fin;
```

```
fonction appartrec (d v: vstat; d inf, sup : entier; d stat : nomstat;
                                            r place : entier): booléen;
var  mil: entier;
début
    si inf = sup alors
    début
        place := inf;
        appartrec := stat = v [inf] .station
    fin
    sinon
    début
        mil := (inf + sup) div 2;
        si stat ≤ v [mil].station alors
            appartrec := appartrec (v, inf, mil, stat, place)
        sinon
            appartrec:= appartrec (v, mil + 1, sup, stat, place)
    fin
fin;
```

1.4. Cette procédure comporte trois parties :
- appel d'une des versions de la procédure **appart** pour déterminer si la station figure déjà dans **v [1 .. nst]**
- si non, insertion dans **v [1 .. nst]** à la **place** délivrée par **appart**
- dans tous les cas, insertion de **noligne** et **ordre** dans la liste correspondante.

```
procédure insertion (dr v: vstat; dr nst: entier; d stat: nomstat;
                                            d noligne, ordre: entier);
 spécification { } => {la station stat, qui se trouve à la place ordre sur la ligne
                                    noligne, a été insérée dans v[1..nst]}
var place, i: entier;
début
    si non appart (v, nst, stat, place) {recherche de stat, calcul de place}
    alors
    début {insertion de stat dans v [ 1 .. nst]
        pour i := nst bas place faire
            v [i + 1] := v [i];
        nst := nst + 1;
        v [place] .station := stat;
        v [place] .adr := nil
    fin;
    insert (v [place].adr, noligne, ordre)
    {insertion de noligne et ordre dans la liste}
fin;
```

1.5. Un simple parcours du fichier, en appelant la procédure **insertion** pour chaque enregistrement, permet de résoudre le problème.

```
procédure consv (dr f : fstat ; dr v : vstat; dr nst : entier) ;
début
    relire (f);
    nst := 0;
    tantque non fdf (f) faire
    début
        insertion (v, nst, f↑.station, f↑.numligne, f↑.rang);
        prendre (f)
    fin;
fin;
```

2. EXPLOITATION DE LA STRUCTURE CRÉÉE DANS LA PREMIERE PARTIE

2.1. Il suffit de vérifier, à l'aide de la fonction **appart**, si **stat** appartient bien à la structure, puis si la liste de lignes correspondante comporte plus d'une cellule.

```
fonction corresp (d v : vstat; d nst : entier; d stat : nomstat) : booléen;
spécification { } => {corresp = stat est une station de correspondance}
var  place: entier;
début
    si appart (v, nst, stat, place) alors
        corresp := v [place] .adr↑.suivant ≠ nil
    sinon
        corresp := faux;
fin;
```

2.2. a) Il s'agit ici d'un simple parcours du vecteur **v [1..nst]** , en comptant les stations dont la liste de lignes comporte plus d'une cellule.

```
fonction nbcorr (d v : vstat; d nst : entier) : entier;
spécification { } => {nbcorr = nombre de stations de correspondance}
var  i, nb: entier;
début
    nb := 0;
    pour i := 1 haut nst faire
        si v [i] .adr↑.suivant ≠ nil alors nb := nb + 1;
    nbcorr := nb
fin;
```

2.2. b) Le parcours récursif du vecteur peut se faire en diminuant à chaque appel la borne supérieure du vecteur.

```
fonction nbcorr (d v : vstat; d nst : entier) : entier;
spécification { } => {nbcorr = nombre de stations de correspondance}
début
```

```
     si nst = 0 alors  nbcorr := 0
     sinon
          si v [nst].adr↑.suivant ≠ nil alors
               nbcorr := 1 + nbcorr (v, nst - 1)
          sinon
               nbcorr := nbcorr(v, nst - 1);
fin;
```

2.3. Nous donnons la solution récursive classique (Tome 2, p. 35), adaptée et simplifiée pour délivrer un booléen au lieu d'un pointeur :

```
fonction appartl (d liste : pointeur; d noligne : entier) : booléen;
spécification { } => {appartl =  noligne ∈ liste+}
début
     si liste = nil alors  appartl := faux
     sinon
          si liste↑.numligne ≥ noligne alors
               appartl := liste↑.numligne = noligne
          sinon  appartl := appartl(liste↑.suivant, noligne)
fin;
```

2.4. Il suffit de compter le nombre de stations qui appartiennent à la ligne **noligne**, en utilisant la fonction **appartl**.

```
fonction nbstat (d v : vstat ; d nst, noligne : entier) : entier;
spécification { } => {nbstat = nombre de stations sur la ligne noligne}
var   i, nb: entier;
début
     nb := 0;
     pour i := 1 haut nst faire
          si appartl(v [i].adr, noligne) alors
               nb := nb + 1;
     nbstat := nb
fin;
```

2.5. Cet algorithme se compose de deux parties :
- appel de **appart**, pour vérifier l'appartenance de stat à la structure, et trouver sa place
- appel de **appartl**, pour vérifier l'appartenance de la station à la ligne.

```
fonction présent (d v : vstat ;  d nst : entier; d stat: nomstat;
                                                  d noligne: entier) : booléen;
spécification { } => {présent = stat  appartient à la ligne noligne}
var   place: entier;
début
     si appart (v, nst, stat, place) alors
          présent := appartl (v [place].adr, noligne)
     sinon présent := faux
fin;
```

2.6. Nous allons d'abord écrire une fonction auxiliaire **rech** qui détermine s'il existe un numéro de ligne commun à deux listes :

```
fonction rech (liste1, liste2 : pointeur) : booléen;
début
    si (liste1 = nil) ou (liste2 = nil) alors
        rech := faux
    sinon {liste1 ≠ nil, liste2 ≠ nil}
        si liste1^.numligne < liste2↑.numligne alors
            rech := rech (liste1↑.suivant, liste2)
        sinon
            si liste1↑.numligne > liste2↑.numligne alors
                rech := rech (liste1, liste2↑.suivant)
            sinon
                rech := vrai
fin;
```

Il suffit alors de vérifier l'appartenance des deux stations à la structure, puis d'appeler **rech** avec leurs listes de lignes.

```
fonction mêmeligne (d v : vstat ; d nst : entier; d stat1, stat2 : nomstat)
                                                              : booléen;
spécification { } =>
            {mêmeligne = stat1 et stat2 appartiennent à une même ligne}
var  place1, place2 : entier;
début
    si appart (v, nst , stat1, place1) et appart (v, nst, stat2, place2) alors
        mêmeligne := rech (v [place1].adr, v [place2].adr)
    sinon
        mêmeligne := faux
fin;
```

2.7. Nous cherchons le nombre de lignes. Les lignes sont numérotées de 1 à N, sans trous, par conséquent le nombre de lignes est égal à N, c'est-à-dire au numéro de ligne le plus élevé. Or chaque liste de lignes est triée sur les numéros de ligne, par conséquent le numéro le plus élevé de chaque ligne est le dernier, et N est le maximum des derniers éléments.
Nous allons donc utiliser une fonction auxiliaire qui délivre un pointeur sur le dernier élément de chaque liste (Tome 2, p. 51)

```
fonction dernier (d ligne: pointeur) : pointeur;
spécification {ligne ≠ nil} =>
                    {dernier = pointeur sur la dernière cellule de ligne}
début
    si ligne↑.suivant = nil  alors
        dernier := ligne
    sinon
        dernier := dernier(ligne↑.suivant)
fin;
```

```
fonction nblignes (d v: vstat; d nst: entier) : entier;
spécification { } => {nblignes = nombre de lignes du réseau}
var  i, max : entier;
     der: pointeur;
début
     max := 0;
{initialisation par minorant, car tous les numéros de lignes sont positifs}
     pour i := 1 haut nst faire
     début
          der := dernier (v [i].adr);
          {par construction, aucune liste de lignes n'est vide}
          si max < der↑.numligne alors max := der↑.numligne
     fin;
     nbligne := max
fin;
```

3. CRÉATION D'UNE NOUVELLE STRUCTURE

3.1. Nous allons utiliser une structure auxiliaire : un vecteur **vligne** d'entiers, qui contiendra les numéros (indices dans **v [1..nst]**) des stations de la ligne **noligne**, rangées dans l'ordre de leur rang sur la ligne. Il suffira ensuite de transformer ce vecteur auxiliaire en une liste, en remplaçant les numéros de station par leur nom et l'indicateur de correspondance.

Pour cela, nous utiliserons une fonction auxiliaire **pointl** qui délivre un pointeur sur la cellule contenant **noligne** dans une liste de lignes :

```
fonction pointl (d liste : pointeur; d noligne : entier) : pointeur;
début
     si liste = nil alors pointl := nil
     sinon
          si liste↑.numligne > noligne alors pointl := nil
          sinon
               si liste↑.numligne = noligne alors pointl := liste
               sinon
                    pointl := pointl (liste↑.suivant, noligne)
 fin;
```

puis une procédure **instête2** d'insertion en tête d'une liste de type pointeur2 :

```
procédure instête2 (dr l : pointeur2; d stat : nomstat; d cor : booléen);
var p : pointeur2;
début
     nouveau (p);
     p↑. station := stat ; p↑. cor := cor;
     p↑. suivant := l; l := p
fin;
```

La procédure demandée s'écrit alors :

```
procédure consl (d v : vstat ; d nst, noligne : entier; r ligne : pointeur2);
var   stat : nomstat;
      vligne : tableau [1..100] de entier;
      i, j, rangmax : entier;
      l : pointeur;
      l1 : pointeur2;
début
      rangmax := 0; {initialisation du rang maximum par un minorant}
      pour i := 1 haut nst faire
      début
            l :=pointl (v[i] . adr, noligne);
            si l ≠ nil {la station numéro i  de v appartient à noligne} alors
            début
                  vligne  [l↑.rang] := i;
            {rangement du numéro de la  station à son rang dans  vligne}
                  si l↑.rang > rangmax alors  {mise à jour du rang maximum}
                        rangmax := l↑.rang
            fin;
      fin;

      {création, de droite à gauche, de la liste des stations :}
      l1 := nil;
      pour j := rangmax bas 1 faire
      début
            stat := v [vligne  [j] ] . station;
            {stat = nom de la jème station  dans vligne}
            instête2 (l1, stat, corresp (v, nst, stat))
      fin;
      ligne := l1
fin;
```

3.2. Il suffit d'appliquer la procédure définie ci-dessus à chacune des lignes du réseau :

```
procédure conslignes (d v : vstat; d nst : entier; r lignes : vectligne;
                                                              r n : entier);
var   i: entier;
début
      n := nblignes (v, nst);
      pour i := 1 haut n faire
            consl (v, nst, i, lignes [i])
fin;
```

7
LES ARBRES

7.1. Gestion d'un dictionnaire arborescent

Énoncé

Cette technique est souvent utilisée pour manipuler de gros dictionnaires en minimisant la place occupée. On rappelle, ci-après, le principe de cette organisation.

Supposons que nous ayons les mots suivants :

ART	BAL	COU	COUVE	etc...
ARTICLE	BAR	COUR	COUVENT	
ARTISTE	BARBE	COUTEAU	COUVER	
	BARRE			

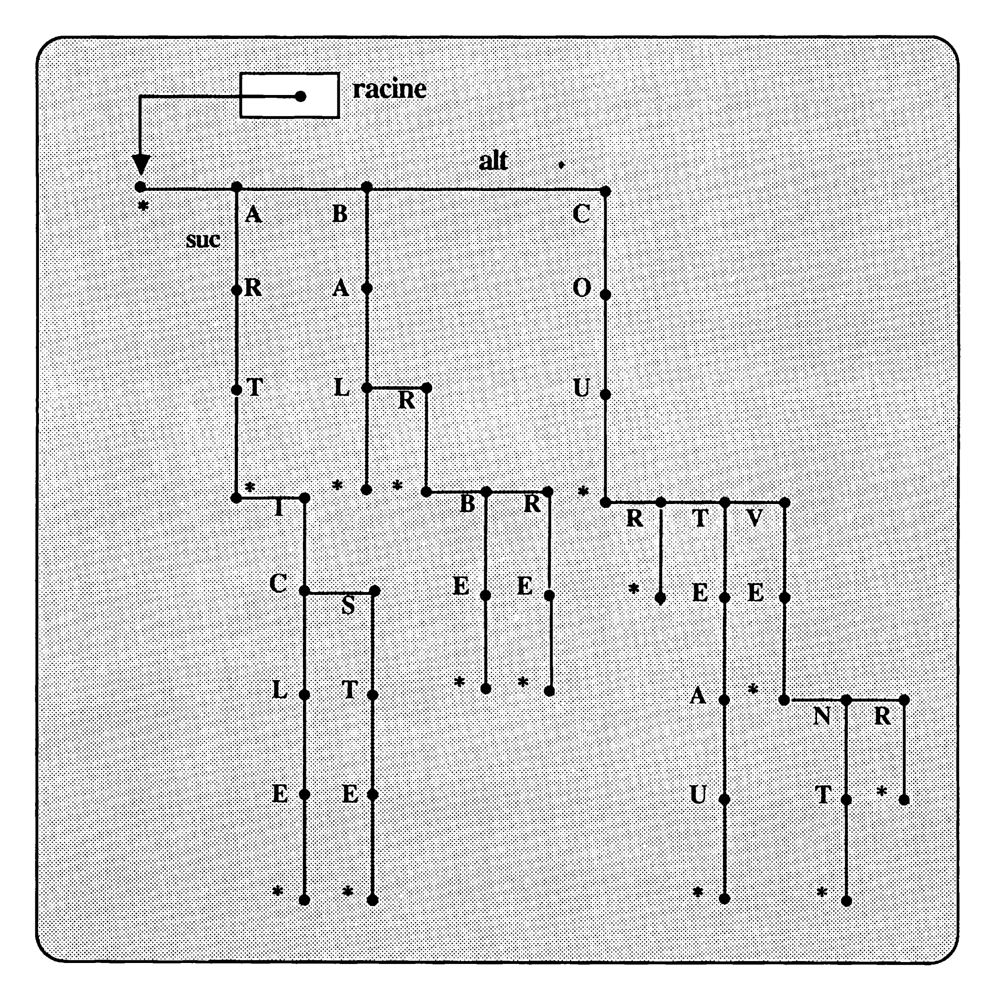
On veut construire, à partir des mots précédents, un dictionnaire ordonné par ordre alphabétique qui tienne compte d'une mise en facteur des préfixes communs.

Par exemple : ART, ARTICLE, ARTISTE ont un préfixe commun ART que l'on peut mettre en facteur.

On admettra que le caractère '*' n'est jamais un caractère entrant dans la composition d'un mot et qu'il est inférieur à tous les autres caractères. On peut alors, afin de simplifier les algorithmes de mise à jour, l'utiliser comme sentinelle : en tête de la structure et comme terminateur de mot.

Chaque information de la forêt est constituée d'un caractère.

On obtient alors la structure suivante :



Les informations associées aux alternants sont ordonnées par ordre alphabétique. Chaque mot est associé à un chemin dans l'arbre. On dispose d'une chaîne de caractères **mot** terminée par le caractére '*'.

1. Écrire, **sous forme récursive**, un algorithme permettant de lister, par ordre alphabétique, les mots du dictionnaire arborescent.

2. Écrire, **sous forme récursive**, un algorithme d'insertion de la chaîne de caractères **mot** dans le dictionnaire arborescent. Cet algorithme n'a aucun effet si la chaîne de caractères est déjà présente dans le dictionnaire.

3. Écrire, **sous forme récursive**, un algorithme de suppression de la chaîne de caractères **mot** dans le dictionnaire arborescent. Cet algorithme n'a aucun effet si la chaîne de caractères est absente du dictionnaire.

Solutions proposées

1. On souhaite écrire, sous forme récursive, une
procédure liste (d racine : pointeur; d mot : chaîne);
spécification {**racine**$^+$ **ordonné**} => {**les mots de racine**$^+$ **ont été imprimés par ordre alphabétique**}
où **mot** est une chaîne de caractères construite au fur et à mesure que l'on effectue un **parcours préfixé** des éléments de **racine**$^+$. On utilise la fonction **concat** de concaténation de deux chaînes de caractères.
Par application directe du cours, l'algorithme est alors le suivant :

```
procédure liste (d racine : pointeur; d mot : chaîne);
spécification {racine+ ordonné} => {les mots de racine+ ont été imprimés
                                                par ordre alphabétique}
début
    si racine ≠ nil alors
    début
        si racine↑.info = '*' alors
            écrireln (mot)
        sinon
            liste (racine↑.suc , concat (mot , racine↑.info));
        liste (racine↑.alt , mot) ;
    fin ;
fin ;
```

Remarque : l'appel de liste doit se faire avec **mot** = ''.

2. On veut écrire, **sous forme récursive**, une
procédure insérer (dr racine : pointeur; d mot : chaîne; d i : entier);
spécification {**racine**$^+$ **ordonné**} => {**mot appartient au dictionnaire ordonné**}
Il faut effectuer un parcours en parallèle de **mot** et de l'arborescence **racine**$^+$ afin de chercher le préfixe commun. Le préfixe commun ayant été trouvé, on insère ensuite, s'il n'est pas vide, le suffixe restant dans mot. Cet algorithme se décompose alors en deux parties : la recherche du préfixe commun suivie par l'appel d'une procédure **suffixe** d'insertion du suffixe.

Hypothèse : Les i-1 premiers caractères de mot appartiennent au dictionnaire
Les cas sont alors les suivants :
. racine = nil
Le parcours de l'arbre est terminé. Le préfixe commun a été trouvé, on insère alors le suffixe restant en exécutant l'action :
suffixe (racine , mot , i) ;
. racine ≠ nil
on a encore 3 cas :

.. racine↑.info > mot [i]

Le parcours de l'arbre est terminé. Le préfixe commun a été trouvé, on insère alors le suffixe restant en exécutant l'action :

suffixe (racine , mot , i) ;

.. racine↑.info < mot [i]

si le caractère mot [i] est présent dans le dictionnaire, il se trouve en position alternant. L'algorithme se poursuit alors par l'appel :

insérer (racine↑.alt , mot , i)

.. racine↑.info = mot [i]

le parcours doit se poursuivre avec le successeur de racine et de i. On effectue alors l'appel récursif suivant :

insérer (racine↑.suc , mot , i + 1)

L'algorithme est alors le suivant :

```
procédure insérer (dr racine : pointeur; d mot : chaîne; d i : entier);
spécification {racine+ ordonné} => {mot appartient au dictionnaire
ordonné}
début
    si racine = nil alors
        {on a trouvé le préfixe, insertion du suffixe}
        suffixe (racine , mot , i)
    sinon
        si racine↑.info > mot [ i ] alors
            {on a trouvé le préfixe, insertion du suffixe}
            suffixe (racine , mot , i)
        sinon
            si racine↑.info < mot [ i ] alors
                insérer (racine↑.alt, mot, i)
            sinon
                insérer (racine↑.suc, mot, i + 1)
fin;
```

Il faut maintenant écrire une

```
procédure suffixe (dr racine : pointeur; d mot : chaîne; d i : entier);
spécification {mot [ 1..i-1 ] ∈ racine-, racine+ ordonné} =>
                                  {mot appartient au dictionnaire ordonné}
```

Cette procédure se décompose en deux parties :

insertion du caractère mot [i] suivie de l'insertion des caractères suivants réalisée au moyen d'une boucle itérative qui s'arrête après l'insertion du caractère '*'.

L'algorithme est alors le suivant :

procédure suffixe (dr racine : pointeur; d mot : chaîne; d i : entier);
spécification {**mot [1..i-1] ∈ racine⁻ , racine⁺ ordonné**} =>
{**mot appartient au dictionnaire ordonné**}

```
var p , q : pointeur;
début
    {insertion de mot [ i ]}
    nouveau (p);
    p↑.info := mot [ i ];
    p↑.alt := racine;
    racine := p;
    {insertion des caractères restants}
    tantque mot [i] ≠ '*'  faire
    début
        i := i + 1 ;
        nouveau (q) ;
        q↑.info := mot [i] ;
        q↑.alt := nil ;
        p↑.suc := q ;
        p := q ;
    fin ;
    p↑.suc := nil ;
fin;
```

4.

On désire écrire, **sous forme récursive**, une
procédure supprime (d mot : chaîne; d i : entier; dr racine : pointeur;
dr succès , fait : booléen);

Il faut :

Effectuer un parcours en parallèle de **mot** et de l'arborescence **racine⁺** afin de détecter la présence du mot à supprimer. Si le mot n'est pas présent, on ne fait rien ; s'il est présent, il faut supprimer les éléments non communs à d'autres dans l'arborescence. Pour cela, on supprime les éléments en remontant jusqu'à trouver un élément appartenant également à une autre chaîne de caractères. Il faut alors disposer de deux indicateurs : **succès** détectant la présence du mot, et **fait** pour la fin des suppressions.
Afin de simplifier l'algorithme de suppression, on dispose d'une sentinelle('*') en tête de l'arborescence **racine⁺**.

Le parcours en parallèle est effectué, comme précédemment, par des appels récursifs. Ensuite, si l'élément est présent et qu'il faut encore supprimer des éléments, on supprime l'élément correspondant. A l'appel, **succès** et **fait** sont initialisés à la valeur **faux**.

On donne le raisonnement après les appels récursifs :

après exécution de supprime (mot , i + 1 , racine↑.suc , succès , fait) on a les cas suivants :

. succès et non fait

mot est **présent** dans le dictionnaire et il faut encore **supprimer racine↑** on préserve racine par l'action :

p := racine;

ensuite on a deux cas :

.. racine↑.alt ≠ nil

c'est la dernière suppression car il y a un alternant, on met à jour **racine, fait** et on **supprime la cellule :**

fait := vrai;
racine := racine↑.alt;
laisser(p);

.. racine↑.alt = nil

Il n'y a pas d'alternant, on supprime l'élément, **fait** reste toujours à faux.

laisser(p);

. ¬succès ou fait

le mot n'est pas présent dans **racine+** ou les suppressions sont terminées. L'algorithme est alors terminé.

après exécution de supprime (mot , i , racine↑.alt , succès , fait) on a les cas suivants :

. succès et non fait

Les éléments ont déjà été supprimés, il suffit de mettre à jour **racine** et **fait** en exécutant les actions :

fait := vrai;
racine↑.alt := racine↑.alt↑.alt;

. ¬succès ou fait

le mot n'est pas présent dans **racine+** ou les suppressions sont terminées. L'algorithme est alors terminé.

L'algorithme est alors le suivant :

procédure supprime (d mot : chaîne; d i : entier; dr racine : pointeur;
dr succès , fait : booléen);
spécification {racine+ ordonné} => {racine+ ordonné ,
(succès , mot a été supprimée de racine+)
∨ (¬succès, mot n'était pas présent}
var p : pointeur;
début *{ne fait rien si l'élément n'appartient pas}*
si racine ≠ nil alors

```
    début
        si mot [ i ] = racine↑.info alors
        début
            si mot [ i ] = '*' alors
                succès := vrai
            sinon
                supprime(mot , i + 1 , racine↑.suc , succès , fait);
            si succès et non fait alors
            début
                p := racine;
                si racine↑.alt ≠ nil alors
                début
                    fait := vrai;
                    racine := racine↑.alt;
                fin;
                laisser(p);
            fin
        fin
        sinon
            si mot [ i ] > racine↑.info alors
            début
                supprime(mot , i , racine↑.alt , succès , fait);
                si succès et non fait alors
                début
                    fait := vrai;
                    racine↑.alt := racine↑.alt↑.alt;
                fin
            fin;
    fin;
fin;
```

7.2. Plan d'un document

Énoncé

On veut pouvoir traiter et manipuler le plan d'un document sous la forme d'un arbre n-aire.

Exemple:

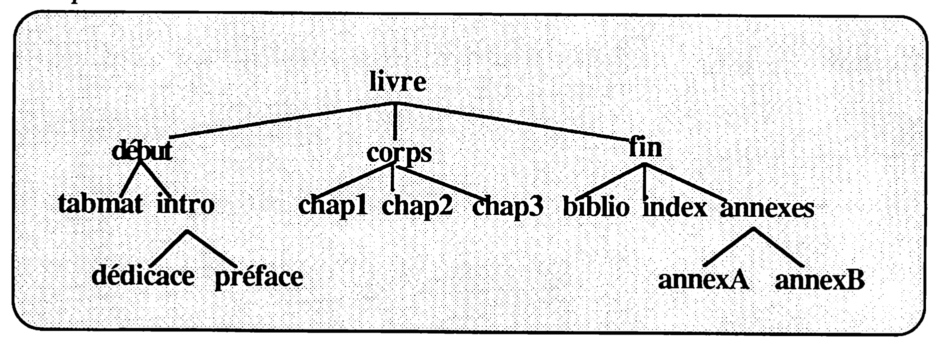


Figure 1

La valeur maximale de n n'est pas connue a priori, et elle peut être assez élevée. Comme on l'a vu dans le Tome 2 (p. 265), on va donc transformer cet arbre en un arbre binaire "alternant-successeur", avec une représentation chaînée (cf Fig. 2).
Les noeuds de l'arbre seront appelés **parties**, chaque partie sera identifiée par son **nom**.

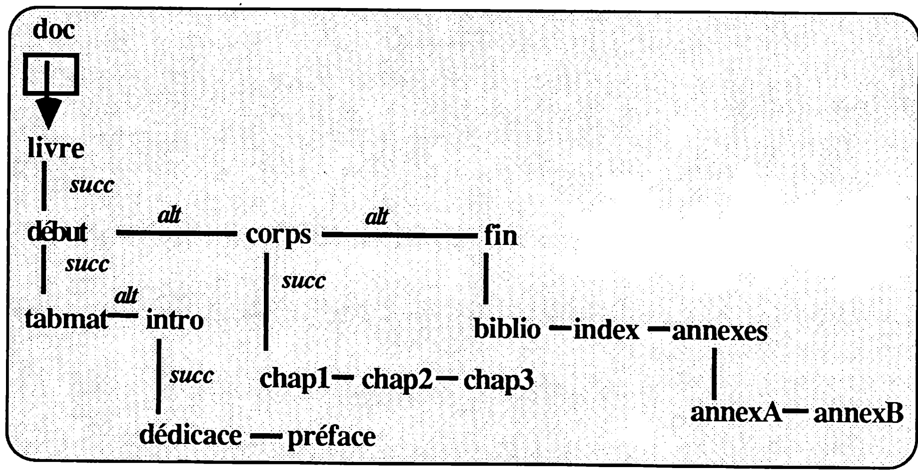


Figure 2

Remarques :

- On supposera que toutes les parties d'un arbre ont un nom différent.

- Le noeud racine de l'arbre (*livre* dans l'exemple) n'a pas d'alternants.

- On dira que A est un **composant** de B si A est soit le successeur direct de B, soit l'un des alternants de ce successeur. Ainsi, *index* est un composant de *fin*, alors que *annexA* ne l'est pas. Par contre, *annexA* est un composant de *annexes*.

- On nommera **feuilles** les parties sans composants. Il faut noter que cette définition d'une feuille est différente de celle qui est habituelle avec les arbres binaires : il s'agit des parties qui n'ont pas de successeur, mais elles peuvent avoir un alternant. Ainsi, *dédicace*, *préface*, *index* sont des feuilles, alors que *intro* et *annexes* ne le sont pas. Bien entendu, il s'agit de retrouver la notion de feuille telle qu'elle se présentait sur l'arbre n-aire initial.

On utilisera les déclarations suivantes :

```
type pointeur  =   ↑partie ;
     partie    =   structure
                        nom : chaîne20 ;
                        alt, succ : pointeur
                   fin ;
```

1. RECHERCHES DE PARTIES DANS L'ARBRE

1.1. Écrire, sous forme itérative puis récursive, une
fonction cherchcomp (d p : pointeur ; d nom : chaîne20) : pointeur;
spécification { } => {**cherchcomp** = pointeur sur la partie **nom** dans la liste de composants **p+**, ou bien nil si **nom** ∉ **p+**}

Exemples : Si p contient l'adresse de la partie *biblio*, alors
cherchcomp (p, 'annexes') délivrera le pointeur sur la partie *annexes*;
cherchcomp (p, 'annexA') ou cherchcomp (p, 'début') délivreront **nil**.

1.2. Écrire une
fonction cherchpartie (d doc : pointeur ; d nom : chaîne20) : pointeur ;
spécification { } => {**cherchpartie** = pointeur sur la partie **nom** dans l'arbre d'adresse **doc**, ou bien nil si **nom** ∉ **doc+**}

a) Algorithme récursif ,
b) Algorithme "semi-récursif", en tenant compte des rôles différents des pointeurs "alt" et "succ".

1.3. Écrire une
fonction comp (d doc : pointeur ; d nom1, nom2 : chaîne20) : pointeur;
spécification { } => {**comp** = pointeur sur le composant **nom2** de la partie **nom1** dans l'arbre d'adresse **doc**, ou bien **nil** si **nom2** n'est pas un composant de **nom1**}

Exemple : comp (doc, 'fin', 'index') délivrera le pointeur sur la partie *index*, alors que comp (doc, 'fin', 'annexB') ou comp (doc, 'fin', 'intro') délivreront **nil**.

1.4. Écrire une
fonction père (d doc : pointeur ; d nom : chaîne20) : pointeur ;
spécification { } => {**père** = adresse de la partie dont **nom** est un composant, dans l'arbre d'adresse **doc**, ou bien **nil** si **nom** $\notin$ **doc**↑.succ+}
Exemples : père (doc, 'chap2') délivrera le pointeur sur *corps*;
père (doc, 'livre') ou père (doc, 'conclusion') délivreront **nil**.

2. COMPTAGE DE PARTIES

2.1. Écrire, sous forme itérative puis récursive, une
fonction nbcomp (d p : pointeur) : entier;
spécification { } => {**nbcomp** = nombre de composants de la liste **p**+}
Exemple : Si p contient l'adresse de la partie *chap1*, alors nbcomp (p) = 3.

2.2. Écrire une
fonction nbcomp2 (d doc : pointeur ; d nom : chaîne20) : entier;
spécification { } => {**nbcomp2** = nombre de composants de la partie **nom** dans **doc**+}
Exemple **:** nbcomp2 (doc, 'corps') = 3.

2.3. Écrire une
fonction nbnoeuds (d p : pointeur) : entier ;
spécification { } => {**nbnoeuds** = nombre de parties dans p+, en comptant p lui-même, ses successeurs et ses alternants}

Exemple : Si p contient l'adresse de la partie *corps*, alors nbnoeuds(p) = 10.

2.4. Écrire une
fonction nbéléments (d p : pointeur) : entier ;
spécification { } =>{**nbéléments** = nombre de sous-parties de la partie d'adresse p}
Exemples : Si p contient l'adresse de la partie *corps*, alors nbnoeuds(p) = 3 ;
si p contient l'adresse de la partie *fin*, alors nbnoeuds(p) = 5.

2.5. Écrire une
fonction nbparties (d doc : pointeur ; d nom : chaîne20) : entier ;
spécification { } =>{**nbparties** = nombre de sous-parties de la partie **nom** dans **doc+**}
Exemples : nbparties(doc, 'corps') = 3 ; nbparties(doc, 'fin') = 5.

3. INSERTION D'UNE NOUVELLE PARTIE

3.1. Écrire une
procédure insertête(dr p : pointeur ; d nom : chaîne20) ;
spécification { } => {insertion d'une nouvelle partie **nom** , sans composants, au début de la liste d'adresse **p**}
Exemple : si p contient l'adresse de *biblio*, alors après exécution de insertête (p, 'conclusion'), on aura p+ =*(conclusion, biblio, index, annexes).*

3.2. On décide de changer le plan de l'ouvrage, en créant une nouvelle partie. Écrire une
procédure insertion (d doc : pointeur ; d nom1, nom2, nom3 : chaîne20);
qui insère la nouvelle partie **nom3** comme composant de **nom1**, et comme alternant immédiat de **nom2** ; si **nom2** est la chaîne vide, c'est que **nom3** devra être en tête de la liste des composants de **nom1**.
La procédure affichera les messages d'erreur appropriés s'il y a erreur sur **nom1**, **nom2**, ou **nom3**.

Pour écrire cette procédure, il est conseillé d'utiliser **insertête**, **cherchcomp** et **cherchpartie**.

4. SUPPRESSION D'ÉLÉMENTS DANS L'ARBRE

4.1. Écrire une
procédure libèretous (d p : pointeur);
spécification { } => {les cellules de l'arbre d'adresse **p** ont été libérées en tenant compte des successeurs et des alternants}

Exemple : si p est un pointeur sur *corps*, les parties *corps* et *fin* et toutes leurs sous-parties seront libérées.

4.2. Écrire une
procédure libère (d p : pointeur) ;
spécification { } =>{toutes les cellules du sous-arbre **p+** ont été libérées , soit **p** et toutes ses parties}

Exemple : si p est un pointeur sur *corps*, seules les parties *corps* , *chap1*, *chap2* et *chap3* seront libérées.

4.3. Écrire une
procédure supp (dr p : pointeur) ;
spécification { } => {le sous-arbre **p+** a été supprimé}
a) Algorithme direct, semi-récursif,
b) Algorithme utilisant "libèretous".

4.4. Écrire une
procédure suppnom (dr p : pointeur ; d nom : chaîne20) ;
spécification { } => {si **nom** appartient à la liste de composants d'adresse **p**, alors la procédure effectue la suppression de la partie **nom** et de tout le sous-arbre correspondant (en libérant les cellules correspondantes) ; sinon la procédure ne fait rien}

Exemple : si on veut supprimer *corps*, p sera le pointeur sur *début*, et on supprimera *corps, chap1, chap2* et *chap3*. L'alternant de *début* deviendra alors *fin*.

4.5. Écrire une
procédure suppression (dr doc : pointeur ; d nom: chaîne20 ; r possible : booléen) ;
spécification { } => {suppression dans l'arbre **doc+** de la partie **nom** et de tout son sous-arbre (en libérant les cellules correspondantes) , **possible** = la suppression a pu être effectuée}
Pour écrire cette procédure, il est conseillé d'utiliser **père, supp et suppnom.**

5. LISTAGES

5.1. Écrire une
procédure listefeuilles (d doc : pointeur) ;
spécification { }=>{les noms de toutes les feuilles de **doc+ ont été imprimés**}

5.2. Écrire, sous forme itérative puis récursive, une
procédure listecomp (d p : pointeur) ;
spécification { } => {impression des noms des composants de la liste **p+**, suivis d'une parenthèse droite}

5.3. Écrire, sous forme semi-récursive puis récursive, une
procédure listelem (d p : pointeur) ;
spécification { } => {impression de l'arbre **p+** ; si **p** vaut **nil** ou n'a pas de composants, la procédure ne fait rien}

Le contenu de l'arbre sera imprimé de la manière suivante : pour chaque partie qui n'est pas une feuille, on imprimera son nom, suivi de la liste de ses composants entre parenthèses.

Exemple :

livre = (début corps fin)
début = (tabmat intro)
intro = (dédicace préface)
corps = (chap1 chap2 chap3)
fin = (biblio index annexes)
annexes = (annexA annexB)

5.4. Écrire une
procédure listplan (d doc : pointeur) ;
spécification { } => {impression du plan complet du document **doc+**, en prévoyant des cas particuliers pour le document vide et le document sans composants}

La présentation du plan, dans le cas général, sera celle qui a été proposée pour la procédure **listelem**.

Solutions proposées

1. RECHERCHES DE PARTIES DANS L'ARBRE

1.1. Pour ces deux fonctions, il s'agit de reprendre les algorithmes classiques de recherche d'un élément dans une liste linéaire chaînée (cf Tome 2, p. 30 à 33) . C'est le pointeur **alt** qui permet d'accéder à l'élément suivant de la liste de composants.

Algorithme itératif :

```
fonction cherchcomp (d p : pointeur ; d nom : chaîne20) : pointeur;
spécification { } => {cherchcomp = pointeur sur la partie nom dans la liste
                              de composants p+, ou bien nil si nom ∉ p+}
var trouvé : booléen;
début
    trouvé := faux;
    tantque (p ≠ nil) et non trouvé faire
        si p↑.nom = nom alors
            trouvé := vrai
        sinon
            p := p↑.alt;
    cherchcomp := p
fin;
```

Algorithme récursif :

```
fonction cherchcomp (d p : pointeur ; d nom : chaîne20) : pointeur;
spécification { } => {cherchcomp = pointeur sur la partie nom dans la liste
                              de composants p+, ou bien nil si nom ∉ p+}
début
    si p = nil alors
        cherchcomp := nil
    sinon
        si p↑.nom = nom alors
            cherchcomp := p
        sinon
            cherchcomp := cherchcomp(p↑.alt, nom)
fin;
```

1.2.

a) C'est l'algorithme classique de recherche d'un élément dans un arbre binaire, les pointeurs "succ" et "alt" jouant les rôles de "gauche" et "droite". Remarque : on a choisi ici d'effectuer une recherche "**en profondeur**", puisqu'on utilise ici d'abord le pointeur "succ", et ensuite seulement, si la recherche n'a pas abouti, le pointeur "alt". Pour obtenir une recherche "**en largeur**", il suffirait d'inverser cet ordre. Mais, avec l'hypothèse que toutes les parties ont des noms différents, le résultat sera le même dans les deux cas. Seul le temps d'accès pourra différer.

```
fonction cherchpartie (d doc : pointeur ; d nom : chaîne20) : pointeur ;
spécification { } => {cherchpartie = pointeur sur la partie nom dans l'arbre
                              d'adresse doc, ou bien nil si nom ∉ doc+}
var p : pointeur;
début
    si doc = nil alors
        cherchpartie := nil
    sinon
        si doc↑.nom = nom alors
            cherchpartie := doc
        sinon
        début
            p := cherchpartie(doc↑.succ, nom);
            si p ≠ nil alors
                cherchpartie := p
            sinon
                cherchpartie := cherchpartie(doc↑.alt, nom)
        fin
fin;
```

1.2. b)

```
fonction cherchpartie (d doc : pointeur ; d nom : chaîne20) : pointeur ;
spécification { } => {cherchpartie = pointeur sur la partie nom dans l'arbre
                              d'adresse doc, ou bien nil si nom ∉ doc+}
var  p : pointeur;  trouvé : booléen;
début
    si doc = nil alors
        cherchpartie := nil
    sinon
        si doc↑.nom = nom alors
            cherchpartie := doc
        sinon
        début
            doc := doc↑.succ ;
            {recherche récursive parmi les composants de doc ,
                              parcourus par une boucle itérative}
            trouvé := faux;
            p := nil;
            tantque (doc ≠ nil) et non trouvé faire
            début
                p := cherchpartie(doc, nom);
                si p ≠ nil alors
                    trouvé := vrai
                sinon
                    doc := doc↑.alt
            fin;
            cherchpartie := p
        fin
fin;
```

1.3.

Il s'agit ici d'une simple utilisation successive des deux algorithmes précédents.

```
fonction comp (d doc : pointeur ; d nom1, nom2 : chaîne20) : pointeur;
spécification { } => {comp = pointeur sur le composant nom2 de la partie
                              nom1 dans l'arbre d'adresse doc,
                  ou bien nil si nom2 n'est pas un composant de  nom1}
var p : pointeur;
début
    p := cherchpartie (doc, nom1);
    comp := cherchcomp (p↑.succ, nom2)
fin;
```

1.4.

```
fonction père (d doc : pointeur ; d nom : chaîne20) : pointeur ;
spécification { } => {père = adresse de la partie dont nom est un
                              composant, dans l'arbre d'adresse doc,
                                       ou bien nil si nom ∉ doc↑.succ+}
var p : pointeur;
début
    si doc = nil alors
        père := nil
    sinon
        si cherchcomp(doc↑.succ, nom) ≠ nil alors
            {nom est un composant de doc}
            père := doc
        sinon
        début
            {recherche récursive, en profondeur (succ avant alt)}
            p := père(doc↑.succ, nom);
            si p ≠ nil alors
                père := p
            sinon
                père := père(doc↑.alt, nom)
        fin
fin;
```

2. COMPTAGE DE PARTIES

2.1.

Dans les deux cas, il s'agit de reprendre les algorithmes classiques de calcul du nombre d'éléments d'une liste linéaire chaînée (cf Tome 2, p. 21). Le pointeur "alt" permet d'accéder à l'élément suivant de la liste de composants.

Algorithme itératif :

```
fonction nbcomp (d p : pointeur) : entier;
spécification { } => {nbcomp = nombre de composants de la liste p+}
var n : entier;
début
    n := 0;
    tantque p ≠ nil faire
    début
        n := n + 1;  p := p↑.alt
    fin;
    nbcomp := n
fin;
```

Algorithme récursif :

```
fonction nbcomp (d p : pointeur) : entier;
spécification { } => {nbcomp = nombre de composants de la liste p+}
début
    si p = nil alors
        nbcomp := 0
    sinon
        nbcomp := 1 + nbcomp(p↑.alt);
fin;
```

2.2.

Utilisation de deux fonctions déjà écrites :

```
fonction nbcomp2 (d doc : pointeur ; d nom : chaîne20) : entier;
spécification { } => {nbcomp2 = nombre de composants de la partie nom
                                                          dans doc+}
var p : pointeur;
début
    p := cherchpartie(doc, nom);
    nbcomp2 := nbcomp(p↑.succ)
fin;
```

2.3.

Il s'agit ici de reprendre l'algorithme classique de comptage des noeuds dans un arbre binaire, les pointeurs "succ" et "alt" jouant les rôles de "gauche" et "droite".

```
fonction nbnoeuds (d p : pointeur) : entier ;
spécification { } => {nbnoeuds = nombre de parties dans p+, en comptant p
                          lui-même, ses successeurs et ses alternants}
début
    si p = nil alors
        nbnoeuds := 0
    sinon
        nbnoeuds := 1 + nbnoeuds(p↑.succ) + nbnoeuds(p↑.alt)
fin;
```

2.4.

```
fonction nbéléments (d p : pointeur) : entier ;
spécification { } =>{nbéléments = nombre de sous-parties de la partie
                                                          d'adresse p}
début
    si p = nil alors
        nbéléments := 0
    sinon
        nbéléments := nbnoeuds(p↑.succ)
fin;
```

2.5.
On peut utiliser la fonction intermédiaire **nbéléments :**
fonction nbparties (d doc : pointeur ; d nom : chaîne20) : entier ;
spécification { } =>{**nbparties** = nombre de sous-parties de la partie **nom** dans **doc+**}
```
début
    nbparties := nbéléments (cherchpartie (doc, nom))
fin;
```

ou bien, on peut utiliser directement **nbnoeuds** :
fonction nbparties (d doc : pointeur ; d nom : chaîne20) : entier ;
spécification { } =>{**nbparties** = nombre de sous-parties de la partie **nom** dans **doc+**}
```
var p : pointeur;
début
    p := cherchpartie(doc, nom);
    si p = nil alors
        nbparties := 0
    sinon
        nbparties := nbnoeuds(p↑.succ)
fin;
```

3. INSERTION D'UNE NOUVELLE PARTIE

3.1.
procédure insertête(dr p : pointeur ; d nom : chaîne20) ;
spécification { } => {insertion d'une nouvelle partie **nom** , sans composants, au début de la liste d'adresse **p**}
```
var  cel : pointeur;
 début
    nouveau(cel);
    cel↑.nom := nom;
    cel↑.succ := nil;
    cel↑.alt := p;
    p := cel;
fin;
```

3.2.
Il s'agit ici de combiner judicieusement les recherches de nom1, nom2 et nom3, puis, si cela est possible, d'effectuer l'insertion demandée à l'aide de la procédure **insertête** :

procédure insertion (d doc : pointeur ; d nom1, nom2, nom3 : chaîne20);
```
var p , q , r : pointeur;
début
```

```
      p := cherchpartie(doc, nom1);
      si p = nil {nom1 ∉ doc+} alors écrireln ('père inconnu')
      sinon
      début
            q := cherchpartie(doc, nom3);
            si q ≠ nil {nom3 ∈ doc+}   alors
                  écrireln ('partie existe déjà')
            sinon
                  si nom2 = '' alors insertête(p↑.succ, nom3)
                  sinon
                  début
                        r := cherchcomp(p↑.succ, nom2);
                        si r = nil {nom2 n'est pas un composant de nom1} alors
                              écrireln ('place d''insertion inexistante')
                        sinon insertête(r↑.alt, nom3)
                  fin
      fin
fin;
```

4. SUPPRESSION D'ÉLÉMENTS DANS L'ARBRE

4.1.

Il s'agit d'une procédure récursive qui traite l'arbre comme un arbre binaire classique.

```
procédure libèretous (d p : pointeur);
spécification { } => {les cellules de l'arbre d'adresse p ont été libérées
                     en tenant compte des successeurs et des alternants}
début
      si p ≠ nil alors
      début
            libèretous(p↑.succ);
            libèretous(p↑.alt);
            laisser(p)
      fin
fin;
```

4.2.

Cette manière de procéder permet de préserver les alternants de **p.**

```
procédure libère (d p : pointeur) ;
spécification { } =>{toutes les cellules du sous-arbre p+ ont été libérées ,
                                          soit p et toutes ses parties}
début
      libèretous(p↑.succ);
      laisser(p)
fin;
```

4.3. a)

procédure supp (dr p : pointeur) ;
spécification { } => {le sous-arbre **p+** a été supprimé}
var q : pointeur;
début
    si p ≠ nil alors
    début
        tantque p↑.succ ≠ nil faire
            supp(p↑.succ);
        q := p; p := p↑.alt;
        laisser(q)
    fin
fin;

4.3.b)

procédure supp (dr p : pointeur) ;
spécification { } => {le sous-arbre **p+** a été supprimé}
var q : pointeur;
début
    libèretous(p↑.succ);
    q := p; p := p↑.alt;
    laisser(q);
fin;

4.4.

Cette procédure ressemble à la suppression d'un élément dans une liste (cf Tome 2, p. 70). Mais ici on a remplacé "laisser" par "libère", afin de supprimer tout l'arbre de chaque élément de la liste.

procédure suppnom (dr p : pointeur ; d nom : chaîne20) ;
spécification { } => {si **nom** appartient à la liste de composants d'adresse **p**, alors la procédure effectue la suppression de la partie **nom** et de tout le sous-arbre correspondant (en libérant les cellules correspondantes) ; sinon la procédure ne fait rien}
var q : pointeur;
début
    si p ≠ nil alors
    début
        si p↑.nom = nom alors
        début
            q := p; p := p↑.alt;
            libère(q)
        fin
        sinon
            suppnom(p↑.alt, nom)
    fin
fin;

4.5.

```
procédure suppression (dr doc : pointeur ; d nom: chaîne20 ;
                                            r possible : booléen) ;
spécification { } => {suppression dans l'arbre doc+ de la partie nom et de
          tout son sous-arbre (en libérant les cellules correspondantes) ,
                              possible = la suppression a pu être effectuée}
var p : pointeur;
début
    possible := faux;
    si doc ≠ nil alors
        si doc↑.nom = nom alors
        début
            supp(doc);
            possible := vrai
        fin
        sinon
        début
            p := père(doc, nom);
            si p ≠ nil alors
            début
                suppnom(p↑.succ, nom);
                possible := vrai
            fin
         fin
fin;
```

5. LISTAGES

5.1.

Il s'agit d'une procédure récursive classique sur un arbre binaire, mais la définition des feuilles est différente : il s'agit des noeuds qui n'ont pas de successeur ; cependant ils peuvent avoir un alternant.

```
procédure listefeuilles (d doc : pointeur) ;
spécification { } =>{le nom de toutes les feuilles de doc+ ont été imprimés}
début
    si doc ≠ nil alors
    début
        si doc↑.succ = nil alors
            écrireln(doc↑.nom)
        sinon
            listefeuilles(doc↑.succ);
        listefeuilles(doc↑.alt)
    fin
fin;
```

5.2. Algorithme itératif :

```
procédure listecomp (d p : pointeur) ;
spécification { } => {impression des noms des composants de la liste p+,
                                        suivis d'une parenthèse droite}
début
    tantque p ≠ nil faire
    début
        écrire(p↑.nom, ' ');
        p := p↑.alt
    fin;
    écrireln(')');
fin;
```

Algorithme récursif :

```
procédure listecomp (d p : pointeur) ;
spécification { } => {impression des noms des composants de la liste p+,
                                        suivis d'une parenthèse droite}
début
    si p = nil alors
        écrireln(')')
    sinon
    début
        écrire(p↑.nom, ' ');
        listecomp(p↑.alt)
    fin
fin;
```

5.3. Algorithme semi-récursif :

```
procédure listelem (d p : pointeur) ;
spécification { } => {impression de l'arbre p+;  si p vaut nil ou n'a pas de
                                  composants, la procédure ne fait rien}
var q : pointeur;
début
    si p ≠ nil alors
        si p↑.succ ≠ nil alors
        début {p+ possède des composants}
            écrire (p↑.nom, '=(');
            listecomp (p↑.succ);
            {la parenthèse droite est fournie par listecomp}
            q := p↑.succ;
            tantque q ≠ nil faire
            début
                listelem (q);  q := q↑.alt
            fin
        fin
fin;
```

Algorithme récursif :

procédure listelem (d p : pointeur) ;
spécification { } => {impression de l'arbre **p+**; si **p** vaut **nil** ou n'a pas de composants, la procédure ne fait rien}

```
var q : pointeur;
début
    si p ≠ nil alors
    début
        si p↑.succ ≠ nil alors
        début
            écrire (p↑.nom, '=(');
            listecomp (p↑.succ);
            listelem (p↑.succ)
        fin;
        listelem (p↑.alt)
    fin
fin;
```

5.4.

Il suffit de traiter d'abord les deux cas particuliers, puis d'appeler la procédure **listelem** pour le cas général.

procédure listplan (d doc : pointeur) ;
spécification { } => {impression du plan complet du document **doc+**, en prévoyant des cas particuliers pour le document vide et le document sans composants}

```
var p : pointeur;
début
    si doc = nil alors
        écrireln('structure vide')
    sinon
        si doc↑.succ = nil alors
            écrireln('une seule partie :', doc↑.nom)
        sinon
            listelem (doc);
fin;
```

8

STRUCTURES DIVERSES

Dans les deux premiers exercices ci-dessous, nous présentons deux méthodes de tri un peu différentes de celles qui ont été vues dans le cours, particulièrement performantes pour des fichiers volumineux (taille ou nombre des enregistrements). Ces méthodes utilisent des structures internes de vecteurs et de listes.

Le troisième exercice propose l'étude comparative de différentes structures pour une même réalisation : liste linéaire triée, table alphabétique, table à adressage aléatoire, et enfin arbre binaire ordonné.

8.1. Tri par index

Énoncé

Nous disposons au départ d'un fichier non trié de mots de 5 lettres au plus chacun. On supposera que ce fichier contient moins de 100 mots, afin de pouvoir le copier initialement dans un vecteur de dimension 100.

A l'aide de comparaisons entre éléments de ce vecteur, nous construirons un vecteur dit d'**index**, qui nous permettra de retrouver les éléments du vecteur dans l'ordre croissant. Sans effectuer aucune permutation, nous pourrons alors construire le fichier trié correspondant.

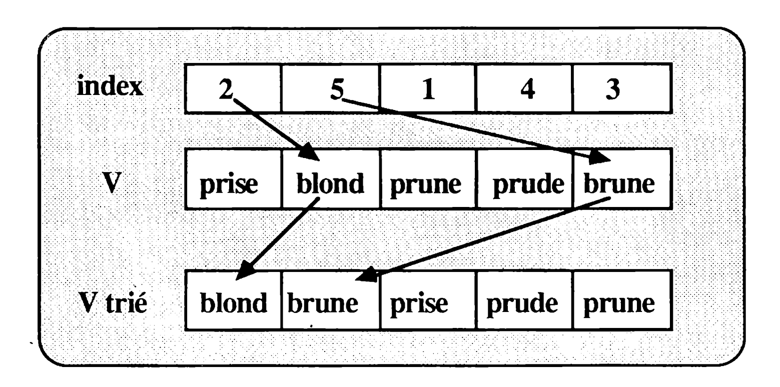
Cette méthode demande une lecture et une écriture de chaque élément du fichier, un nombre de comparaisons d'éléments du vecteur relativement élevé, mais aucun déplacement d'éléments du vecteur. Elle est donc intéressante pour trier, sur une clé simple, des enregistrements de grande taille.

On dispose des types suivants :
type
vectmots = tableau [1..100] de chaîne5;
vectent = tableau [1..100] de entier;
fichmots = fichier de chaîne5;

1. Écrire la procédure
procédure lirefich (dr f: fichmots; r V: vectmots; r n : entier);
spécification {f=$f_1{}^m$, m≤100} =>{**n**=m, **V [1..n] = f**)}

2. Écrire la procédure
procédure créindex (d V:vectmots; d n : entier; r index :vectent)
spécification { } => {création de **index [1..n]** ,
V[index [i]] ≤ V[index [i + 1]], i ∈ [1 .. n-1]}
Exemple:
Si **n**=5 et **V**= *{prise, blond, prune, prude, brune}*
alors **index** = 2 5 1 4 3

En effet :
le plus petit élément de V est *blond,* à l'indice 2, d'où **index** [1]=2
le second élément de V est *brune,* à l'indice 5, d'où **index** [2]=5
....
le plus grand élément de V est *prude,* à l'indice 4, d'où **index** [5]=4



3. Écrire la procédure
procédure écrifichtri (d V : vectmots; d n : entier;
d index : vectent; r f : fichmots);
spécification { }=> {création d'un fichier trié **f** contenant les éléments de
V [1..n] dans l'ordre croissant}

4. En utilisant les procédures précédentes, écrire la procédure
procédure trifich (dr f : fichmots);
spécification {**f**=$f_1{}^m$, m≤100}=> {**f** est trié}

Solutions proposées

1. Il s'agit du parcours simple d'un fichier (Tome 1, p. 65), en comptant les éléments et en les recopiant dans un vecteur. La précondition sur le nombre d'éléments permet d'être sûr que le vecteur a une taille suffisante.

```
procédure lirefich (dr f: fichmots; r V: vectmots; r n : entier);
spécification {f=f1^m, m≤100} => {n=m, V [ 1..n] = f)}
début
    relire(f);
    n := 0;
    tantque non fdf(f) faire
    début
        n := n+1;
        V [ n] := f↑;
        prendre(f)
    fin
fin;
```

2. Première solution: utiliser le vecteur d'indices du tri par comptage (Tome 1, p. 168) :

```
procédure créindex (d V:vectmots; d n : entier; r index :vectent)
spécification { } => {création de index [1..n] ,
                          V[index [i]] ≤ V[index [i + 1]], i ∈ [1 .. n-1]}
var i,j:entier;
    ind :vectent;
début
    {initialisation}
    pour i := 1 haut n faire ind [ i] := 1;
    {construction du vecteur d'indices}
    pour i := 1 haut n-1 faire
        pour j := i+1 haut n faire
            si V [ i] >V [ j] alors
                ind [ i] := ind [ i] +1
            sinon
                ind [ j] := ind [ j]+1;
    {transformation des indices en index}
    pour i := 1 haut n faire index [ind [ i] ] := i
fin;
```

Le nombre de comparaisons entre éléments de V dans cet algorithme est toujours égal à $n(n-1)/2$.

Deuxième solution: tri "bulles optimisé" (Tome 1, p. 160) modifié, en permutant les indices au lieu des éléments du vecteur eux-mêmes:

```
procédure créindex (d V:vectmots; d n : entier; r index :vectent)
spécification { } => {création de index [1..n] ,
                          V[index [i]] ≤ V[index [i + 1]], i ∈ [1 .. n-1]}
```

```
var  i,j:entier;
     perm:booléen;
début
     pour i := 1 haut n faire  index [ i]  := i;
     i := n;  perm := vrai;
     tantque perm faire
     début
          perm := faux;
          pour j := 1 haut i-1 faire {comparaison entre éléments du vecteur)
               si V [ index [ j] ] >V [ index [ j +1] ]  alors
               début
                    {permutation des indices correspondants}
                    permut (index, j, j +1); perm := vrai
               fin;
          i := i-1
     fin
fin;
```

Le nombre de comparaisons entre éléments du vecteur est ici au plus égal à n(n-1)/2, en effet l'optimisation à l'aide du booléen **perm** permet d'arrêter les comparaisons plus tôt s'il y a lieu. Cette solution est donc préférable.

3. Il suffit d'utiliser le vecteur **index** pour obtenir l'indice des éléments que l'on rangera successivement dans le fichier.

```
procédure écrifichtri (d V : vectmots;   d n : entier;
                                         d index : vectent; r  f : fichmots);
spécification { }=> {création d'un fichier trié f contenant les éléments de
                                         V [ 1..n]  dans l'ordre croissant}
début
     récrire(f);
     pour i := 1 haut n faire
     début
          f↑ := V [ index [ i] ] ; mettre(f)
     fin
fin;
```

4. Il suffit d'appliquer successivement les trois procédures décrites ci-dessus.

```
procédure trifich (dr f : fichmots);
spécification {f=f1^m, m≤100}=> {f est trié}
var  V : vectmots;  n : entier;
     index :vectent;
début
     lirefich (f,V,n);
     créindex (V, n, index);
     écrifichtri (V, n, index, f)
fin;
```

8.2. Tri par distribution

Énoncé

La seconde méthode de tri que nous proposons s'applique de manière particulièrement simple à un fichier où tous les mots ont la même longueur : ici, nous avons choisi une longueur de 5 lettres. Elle demande un nombre de lectures et d'écritures du fichier égal à la longueur des mots, soit 5 dans cet exemple. Une petite extension permettrait de traiter les mots plus courts.

Cette méthode n'utilise que la mémoire centrale dynamique, sans recopie du fichier dans un vecteur. Elle permet donc de trier des fichiers de grande taille, la seule limite étant la taille de la mémoire centrale disponible.

Le principe de la méthode consiste à "distribuer" les mots du fichier en sous-classes (ici, des listes chaînées) selon la dernière lettre de chaque mot. Ces sous-classes sont ensuite concaténées pour former un fichier trié sur la dernière lettre. L'opération est alors recommencée pour l'avant-dernière lettre, et ainsi de suite jusqu'à la première lettre.

Exemple :
Fichier initial : *{prise, blond, prune, prude, brune}*
Etape1 (dernière lettre): Sous-classes : 'd' : blond
'e' : prise, prune, prude, brune
fichier : *{blond, prise, prune, prude, brune}*

Etape2 (4ème lettre) : Sous-classes : 'd' : prude
'n' : blond, prune, brune
's' : prise
fichier : *{prude, blond, prune, brune, prise}*

Etape3 (3ème lettre) : Sous-classes : 'i' : prise
'o' : blond
'u' : prude, prune, brune
fichier : *{prise, blond, prude, prune, brune}*

Etape4 (2ème lettre) : Sous-classes : 'l' : blond
'r' : prise, prude, prune, brune
fichier : *{blond, prise, prude, prune, brune}*

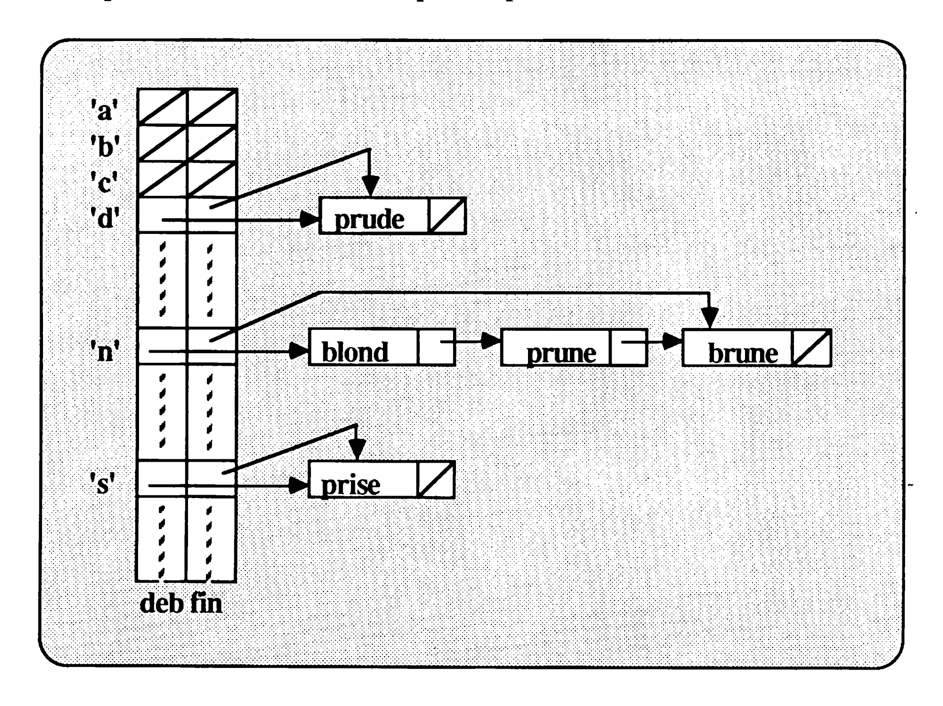
Etape5 (première lettre) : Sous-classes : 'b' : blond, brune
'p' : prise, prude, prune
fichier : *{blond, brune, prise, prude, prune}*

Les types suivants seront utilisés :

type pointeur = ↑**cell;**
cell = **structure**
mot : chaîne5;
suiv : pointeur
fin;
debfin = **structure**
deb,fin:pointeur
fin;
tablistes = **tableau ['a'..'z'] de debfin;**
fichmots = **fichier de chaîne5;**

En effet, les "sous-classes" seront réalisées au moyen de listes chaînées, dont on gardera les adresses de début et de fin dans un tableau indicé sur les lettres de l'alphabet. L'adresse de fin servira à accélérer l'insertion en fin de liste au cours des phases de "distribution", alors que l'adresse de début servira aux phases de concaténation. Bien entendu, toutes ces adresses seront mises à nil au début de chaque étape.

Exemple : les sous-listes de l'étape 2 se présenteront ainsi :



1. Écrire la procédure :
procédure inittablistes (dr tab : tablistes);
spécification { } => {tous les pointeurs de **tab** sont initialisés à **nil**}

2. Écrire la procédure :
procédure insertfinliste (d mot : chaîne5; dr l : debfin);
spécification { } => {**mot** est inséré dans une nouvelle cellule à la fin de la liste d'adresse **l .deb**}

3. Écrire la procédure :
procédure distribue (dr f : fichmots; dr tab : tablistes; d k : entier);
spécification { } => {tous les mots de **f** sont distribués dans les listes dont les adresses figurent dans **tab,** selon la **k**ième lettre}

4. Écrire la procédure :
procédure concatlistes(d tab : tablistes; dr f : fichmots);
spécification { } => {les mots de toutes les listes de **tab** sont écrits dans **f,** dans l'ordre alphabétique}

5. Écrire la procédure :
procédure tri_distrib (dr f : fichmots);
spécification { } => {**f** est trié}

Solutions proposées

1.

```
procédure inittablistes (dr tab : tablistes);
spécification { } => {tous les pointeurs de tab sont initialisés à nil}
var   c : car;
début
      pour c := 'a' haut 'z' faire
      début
            tab [ c] . deb  := nil;
            tab [ c] . fin  := nil
      fin
fin;
```

2. Il faut utiliser le pointeur "fin" pour insérer en fin de liste, mais sans oublier de mettre à jour le pointeur de début de liste ou bien le pointeur de la dernière cellule.

```
procédure insertfinliste (d mot : chaîne5; dr l : debfin);
spécification { } => {mot est inséré dans une nouvelle cellule à la fin
                                        de la liste d'adresse l.deb}
var    p :pointeur;
```

```
début
    nouveau(p);
    p↑.suiv := nil; p↑.mot := mot;
    si l.deb= nil alors
        l.deb := p
    sinon
        l.fin↑.suiv := p;
    l.fin := p;
fin;
```

3. Il s'agit d'un parcours simple du fichier **f**, au cours duquel on applique à chaque mot de **f** la procédure d'insertion en fin de liste, selon la **k**ième lettre, définie ci-dessus.

```
procédure distribue (dr f : fichmots; dr tab : tablistes; d k : entier);
spécification { } => {tous les mots de f sont distribués dans les listes dont
                      les adresses figurent dans tab, selon la kième lettre}
début
    relire(f);
    tantque non fdf(f) faire
    début
        insertfinliste (f↑, tab [ mot [ k] ]);
        prendre(f)
    fin;
fin;
```

4. Parcours successif de toutes les listes dont les adresses figurent dans tab ; l'espace occupé par ces listes est récupéré au fur et à mesure du traitement.

```
procédure concatlistes(d tab : tablistes; dr  f : fichmots);
spécification { } => {les mots de toutes les listes de tab sont écrits dans f,
                      dans l'ordre alphabétique}
var   c:car; l,p:pointeur;
début
    récrire(f);
    pour c := 'a' haut 'z' faire
    début
        l := tab [ c] .deb;
        tantque l ≠ nil faire
        début
            p := l;
            f↑ := l↑. mot;  mettre (f);
            l := l↑.suiv;
            laisser(p)
        fin;
    fin;
fin;
```

5. Pour chacune des cinq lettres des mots, on itère le processus de distribution et concaténation.

```
procédure tri_distrib (dr f : fichmots);
spécification { } => {f est trié}
var   tab : tablistes;
      k:entier;
début
      pour k := 5 bas 1 faire
      début
            inittablistes(tab);
            distribue(f,tab, k);
            concatlistes(tab, f);
      fin
fin;
```

8.3. Index d'un livre

Énoncé

On souhaite construire l'index alphabétique des noms propres cités dans un livre et, pour cela, on se propose de choisir entre plusieurs structures de données.

L'élément de base de chacune des structures sera la **ligne**. Une ligne, dans l'index, sera composée d'un nom propre, associé aux numéros de page où il figure dans le livre. Ces numéros seront chaînés entre eux, dans l'ordre croissant, sans répétition. Deux pointeurs, l'un sur le premier et l'autre sur le dernier, permettront de gérer la liste des numéros. On accèdera aux lignes, et on les reliera entre elles par des pointeurs, de manières différentes selon les structures adoptées.

```
type  pnum   = ↑num;
      num    = structure
                   numpage : entier;
                   numsuiv : pnum
               fin;
      pligne = ↑ligne;
      ligne  = structure
                   nom : chaîne25 ;
                   premier, dernier : pnum;
                   ...      {pointeur(s) sur ligne(s) suivante(s),
                                  selon la structure adoptée}
               fin;
```

Exemple : Napoléon est cité aux pages 10, 147 et 392. La ligne correspondante sera :

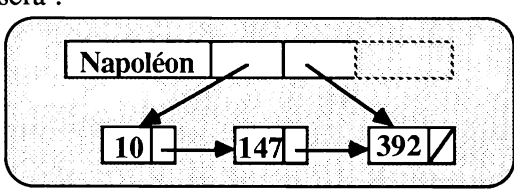


Figure 1

Cette ligne sera imprimée dans l'index sous la forme :
Napoléon : 10, 147, 392

1. PROCÉDURES UTILISÉES DANS TOUTES LES STRUCTURES

1.1. Écrire la
fonction crécellpage (d page : entier) : pnum ;
qui délivre un pointeur sur une nouvelle cellule de type pnum, comportant le numéro **page** .

1.2. Écrire la
procédure inserpage (d l : pligne ; d page : entier) ;
qui insère le numéro **page** à la fin de la liste des numéros associé à la "ligne" d'adresse **l**. Cette liste contient déjà au moins un numéro à l'entrée dans la procédure.

1.3. Écrire la
procédure impligne (d l : pligne) ;
qui imprime toutes les informations d'une ligne : le nom suivi de la liste de numéros de page.

2. LISTE LINÉAIRE

La structure A est une **liste chaînée** de lignes, **triée** sur les noms.

```
type  pligne =  ↑ligne;
      ligne  =  structure
                    nom : chaîne25 ;
                    premier, dernier : pnum;{pnum déclaré ci-dessus}
                    lignesuiv : pligne;
                fin;
var   lind : pligne;
```

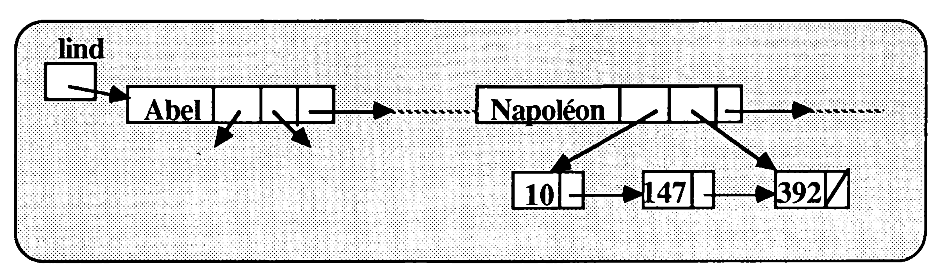


Figure 2

2.1. Écrire la
procédure insnomtête (dr l : pligne; d nom : chaîne25 ; d page : entier) ;
qui insère une nouvelle "ligne" en tête de la liste **l**, avec **nom** et **page** .

2.2. Écrire, sous forme récursive, la
procédure inserta (dr lind : pligne ; d nom : chaîne25; d page : entier) ;
Si **nom** ne figure pas encore dans la liste triée **lind**, la procédure effectue l'insertion de **nom** dans la liste, associé au numéro de page **page**.
Si **nom** figure déjà dans la liste, la procédure vérifie si le dernier numéro de la liste de numéros associée à **nom** est égal à **page**. Si oui, la procédure ne fait rien, si non elle insère **page** à la fin de cette liste de numéros.

3. TABLE DE LISTES LINÉAIRES (ACCES SUR L'INITIALE)

La structure B est une table, indicée sur les lettres majuscules de l'alphabet, de pointeurs sur des **listes chaînées triées** de noms ayant la même initiale. Cette table aura été initialisée avec **nil** dans tous ses éléments.

type tablett = tableau ['A' ..'Z'] de pligne ; *{pligne déclaré ci-dessus}*

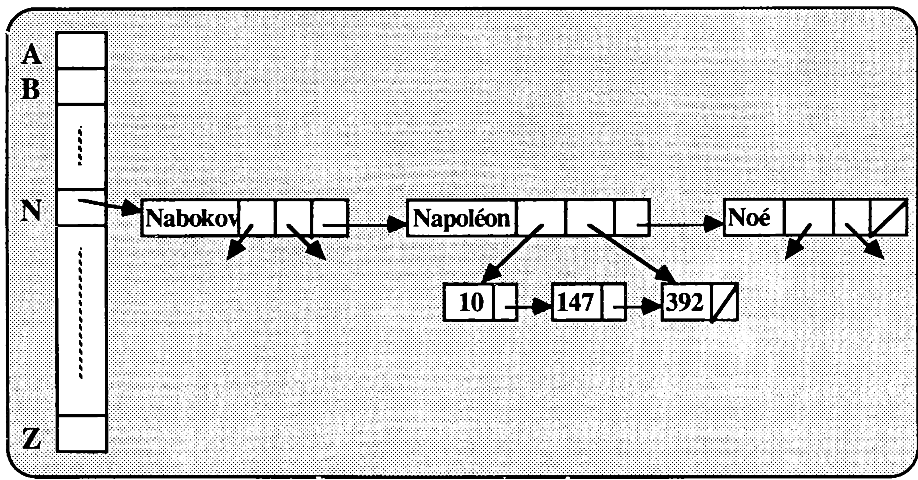


Figure 3

Écrire la
procédure insertb (dr tabl : tablett ; d nom : chaîne25; d page : entier) ;
qui insère **nom** et **page** dans la structure B s'ils n'y figurent pas déjà, comme ci-dessus.

4. TABLE DE LISTES LINÉAIRES (ACCES ALÉATOIRE)

La structure C est une table de 200 pointeurs sur des **listes chaînées triées** de noms ayant le même "code" (cf Figure 4). Ce code est un nombre compris entre 1 et 200.

type indtab = 1..200;
tabcode = tableau [indtab] de pligne ;

Cette table aura été initialisée avec **nil** dans tous ses éléments.

4.1 Écrire la fonction :
fonction code (d nom : chaîne25) : indtab ;
spécification { } =>
{**code** = (somme des codes ascii des lettres de **nom**) mod 200) +1 }

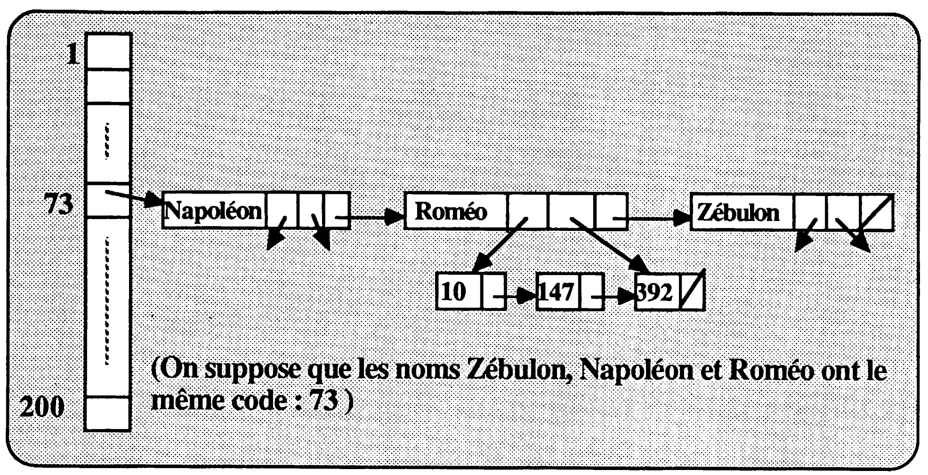


Figure 4

4.2. Écrire la procédure :
procédure insertc (dr tabc : tabcode ; d nom : chaîne25; d page : entier) ;
qui insère **nom** et **page** dans la structure C s'ils n'y figurent pas déjà, comme ci-dessus.

5. ARBRE BINAIRE ORDONNÉ

La structure D est un arbre binaire ordonné sur les noms :

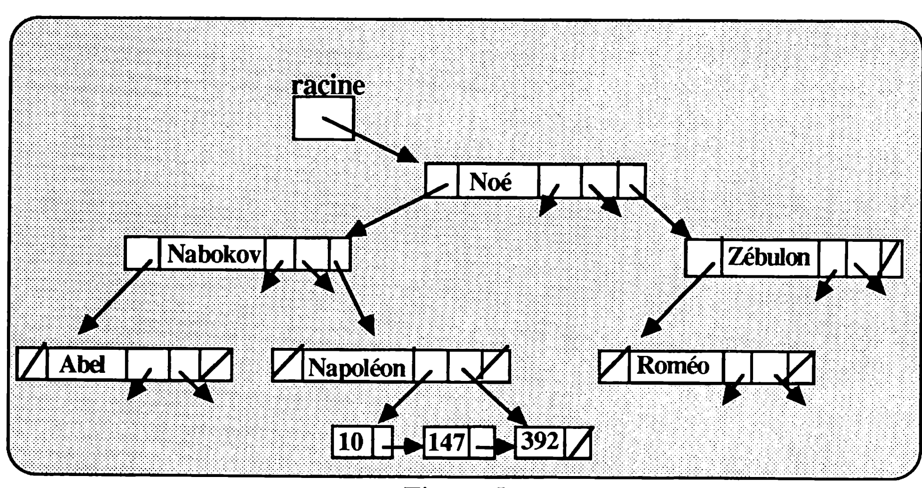


Figure 5

On donne les déclarations suivantes :

type **pligne** = ↑**ligne;**
    ligne = **structure** *{attention, le type ligne a été modifié}*
        gauche : pligne;
        nom : chaîne25 ;
        premier, dernier : pnum; *{pnum n'a pas changé}*
        droite : pligne;
    fin;
var **racine : pligne;**

5.1. Écrire, sous forme itérative puis récursive, une
fonction place (d racine : pligne; d nom : chaîne25) : pligne;
spécification {**racine** ≠ **nil**} => {(**place** = père, (**nom** = père↑.nom)
    v (**nom** < père↑.nom, père↑.gauche = **nil**)
    v (**nom** > père↑.nom, père↑.droite = **nil**)}

5.2. Écrire la
fonction crécelligne (d nom : chaîne25; d page : entier): pligne ;
qui délivre l'adresse d'une nouvelle cellule "feuille" de type pligne, comportant les informations **nom** et **page**.

5.3. Écrire la
procédure insertd (dr racine : pligne ; d nom : chaîne25; d page : entier) ;
qui insère **nom** et **page** dans la structure D s'ils n'y figurent pas déjà, comme ci-dessus.

6. CONCLUSIONS

6.1. Parmi les structures A, B, C et D, lesquelles permettent d'imprimer l'index (trié) du livre ?

6.2. Écrire les procédures correspondantes.

6.3. En supposant que 1000 noms différents sont cités dans le livre, classer les structures A, B, C et D selon le temps d'accès moyen à un nom donné (du temps le plus court au temps le plus long).

6.4. Quelles sont donc, parmi les structures proposées ici, les mieux adaptées à la construction de l'index ?

Solutions proposées

1. PROCÉDURES UTILISÉES DANS TOUTES LES STRUCTURES

Il s'agit ici des primitives de gestion de listes chaînées qui serviront dans toutes les structures étudiées dans la suite de l'exercice.

1.1. On a choisi ici de traiter la création de cellule au moyen d'une fonction plutôt que d'une procédure.

```
fonction crécellpage (d page : entier) : pnum ;
var q: pnum;
début
    nouveau(q);
    q↑.numpage := page;
    q↑.numsuiv := nil;
    crécellpage := q
fin;
```

1.2. Lors de la création d'une cellule "ligne" (procédures **insnomtête** ou **crécelligne** ci-dessous), on insère le premier numéro, qui est donc aussi le dernier. On est alors sûr, ensuite, que le pointeur "dernier" ne sera pas égal à nil.

```
procédure inserpage (d l : pligne ; d page : entier) ;
var  q: pnum;
début
    q := crécellpage(page);
    l↑.dernier↑.numsuiv := q;
    l↑.dernier := q
fin;
```

1.3. On pourrait écrire une procédure récursive, mais la version itérative permet de gérer facilement les ponctuations entre numéros successifs, en traitant séparément le dernier numéro.

```
procédure impligne (d  l : pligne) ;
var p: pnum;
début
    écrire(l↑.nom, ':');
    p := l↑.premier;
    tantque p ≠ l↑.dernier faire
    début
        écrire(p↑.numpage , ',');
        p := p↑.numsuiv;
    fin;
    écrireln(p↑.numpage)
fin;
```

2. LISTE LINÉAIRE

2.1. Cette procédure est une variante plus complexe de la procédure "insertête" du cours (cf Tome 2, p. 46). On y adjoint l'initialisation de la liste des numéros de page .

```
procédure insnomtête (dr l : pligne; d nom : chaîne25 ; d page : entier) ;
var   p: pligne; q: pnum;
début
      nouveau(p);
      p↑.nom := nom;
      q := crécellpage(page);
      p↑.premier := q;  p↑.dernier := q;
      p↑.lignesuiv := l;
      l := p;
fin;
```

2.2. De même, cette procédure est une variante plus complexe de la procédure "insertrié" du cours (cf Tome 2, p. 58). On y a ajouté l'insertion du nouveau numéro de page, à condition qu'il soit différent du dernier numéro entré.

```
procédure inserta (dr lind : pligne ; d nom : chaîne25; d page : entier) ;
début
      si lind = nil alors
            insnomtête(lind, nom, page)
      sinon
            si lind↑.nom > nom alors
                  insnomtête(lind, nom, page)
            sinon
                  si lind↑.nom < nom alors
                        inserta(lind^.lignesuiv, nom, page)
                  sinon
                        si lind↑.dernier↑.numpage ≠ page alors
                              inserpage(lind, page);
fin;
```

3. TABLE DE LISTES LINÉAIRES (ACCES SUR L'INITIALE)

Cette procédure est extrêmement simple : il s'agit de l'insertion dans une liste triée, comme ci-dessus . La seule différence consiste dans le fait qu'on dispose d'une table de têtes de listes, et qu'on sélectionne la tête de liste que l'on va utiliser au moyen de l'initiale du nom.

```
procédure insertb (dr tabl : tablett ; d nom : chaîne25; d page : entier) ;
début
      inserta (tabl [  nom [1] ] , nom, page)
fin;
```

Il faut noter que la table sera mise à jour dans le cas où l'on est amené à insérer en tête de liste. Il serait donc tout à fait erroné d'utiliser une variable pointeur auxiliaire et d'écrire :

```
début
    p := tabl [nom [1] ];
    inserta (p , nom, page)
fin;
```

4. TABLE DE LISTES LINÉAIRES (ACCES ALÉATOIRE)

4.1.

```
fonction code (d nom : chaîne25) : indtab ;
spécification {} =>
        {code = (somme des codes ascii des lettres de nom) mod 200) +1}
var i, s: entier;
début
    s := 0;
    pour i := 1 haut 25 faire
        s := s + ord (nom [ i ]);
    code := (s mod 200) + 1
fin;
```

4.2. On peut faire les mêmes remarques ici que dans la partie 3. La seule différence consiste en la manière de sélectionner la tête de liste.

```
procédure insertc (dr tabc : tabcode ; d nom : chaîne25; d page : entier) ;
var i : indtab;
début
    i := code(nom);
    inserta(tabc[ i ] , nom, page)
fin;
```

5. ARBRE BINAIRE ORDONNÉ

5.1. Version itérative :

```
fonction place (d racine : pligne; d nom : chaîne25) : pligne;
spécification {racine ≠ nil} => {(place = père, (nom = père↑.nom)
                                  v (nom < père↑.nom, père↑.gauche = nil)
                                  v (nom > père↑.nom, père↑.droite = nil)}
var continue : booléen;
début
    continue := vrai ;
```

```
    tantque continue faire
        si (nom < racine↑.nom)  et (racine↑. gauche ≠ nil)  alors
            racine := racine^.gauche
        sinon
            si (nom > racine↑.nom) et (racine↑. droite ≠ nil) alors
                racine:= racine↑.droite
            sinon
                continue := faux;
    place := racine
fin;
```

Version récursive :

```
fonction place (d racine : pligne; d nom : chaîne25) : pligne;
spécification {racine ≠ nil} => {(place = père, (nom = père↑.nom)
                                  v (nom < père↑.nom, père↑.gauche = nil)
                                  v(nom > père↑.nom, père↑.droite = nil)}
début
    si (nom < racine↑.nom)  et (racine↑.gauche ≠ nil)  a lors
        place := place(racine↑.gauche, nom)
    sinon
        si (nom > racine↑.nom) et (racine↑.droite ≠ nil) alors
            place := place(racine↑.droite, nom)
        sinon
            place := racine
fin;
```

5.2.

```
fonction crécelligne (d nom : chaîne25; d page : entier): pligne ;
var  l: pligne;  q: pnum;
début
    nouveau(l);
    l↑.nom := nom;
    q := crécellpage(page);
    l↑.premier := q;
    l↑.dernier := q;
    l↑.gauche := nil;
    l↑.droite := nil;
    crécelligne := l
fin;
```

5.3. Il s'agit ici d'une version un peu plus complexe de l'insertion d'un noeud dans un arbre binaire ordonné (cf Tome 2, p. 235). Nous y ajoutons d'une part le cas où l'arbre serait vide (initialisation), d'autre part la mise à jour d'un noeud existant.

```
procédure insertd (dr racine : pligne ; d nom : chaîne25; d page : entier) ;
var   p: pligne;
début
      si racine = nil alors
            racine := crécelligne (nom, page)
      sinon
      début
            p := place(racine, nom);
            si p↑.nom > nom alors
                  p↑.gauche :=crécelligne (nom, page)
            sinon
                   si p↑.nom < nom alors
                        p↑.droite := crécelligne (nom, page)
                  sinon
                        si p↑.dernier↑.numpage ≠ page alors
                              inserpage(p, page)
      fin
 fin;
```

6.1. La structure **A** (liste linéaire triée) permet l'accès à tous les noms dans l'ordre alphabétique, c'est l'ordre où ils sont rangés dans la liste.

Dans la structure **B**, chaque liste est triée, et ne comporte que les noms qui commencent par une même lettre. Il suffit donc de traiter ces listes dans l'ordre des initiales pour retrouver l'ordre alphabétique demandé.

Par contre, dans la structure **C**, bien que chaque liste individuelle soit triée, il est impossible d'atteindre rapidement un ordre global sur l'ensemble des noms. Il faudrait envisager un interclassement de ces listes, ce qui serait tout à fait déraisonnable.

Enfin, l'arbre binaire ordonné de la structure **D** permet de retrouver les noms dans l'ordre alphabétique ; il suffit de le parcourir dans l'ordre infixé. Finalement, **seule la structure C ne permet pas l'impression d'un index trié.**

6.2.

```
procédure impindexa (d lind: pligne);
début
      si lind ≠ nil alors
      début
            impligne(lind);
            impindexa(lind↑.lignesuiv)
      fin
fin;
```

```
procédure impindexb (d tabl: tablett);
var c: car;
début
     pour c := 'A'  haut  'Z' faire  impindexa (tabl[c])
fin;

procédure impindexd (d racine: pligne);
début
     si racine ≠ nil alors
     début
          impindexd(racine↑.gauche);
          impligne(racine);
          impindexd(racine↑.droite)
     fin
fin
```

6.3. Soit k le temps d'accès moyen à une cellule au moyen de son pointeur, et t_x le temps d'accès à un nom donné dans la structure x.

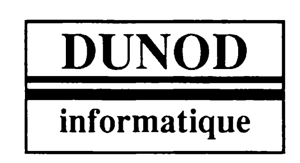
Pour la structure **A**, on parcourt en moyenne la moitié de la liste, donc : **t_A**$\approx$ 1000 k / 2 = **500k** .

Pour la structure **B**, si l'on suppose qu'il y a autant de mots pour chaque initiale, on parcourt en moyenne des listes de 1000/26 $\approx$ 40 éléments, soit, en négligeant le temps d'accès par la table, **t_B**$\approx$ 40 k / 2 $\approx$ **20 k** . En réalité, les listes seront de longueur très variable, et les temps d'accès varieront beaucoup selon l'initiale du nom.

Pour la structure **C**, les listes seront, si la fonction de codage est bien choisie, à peu près de même longueur. De plus, elles seront plus courtes puisque plus nombreuses : 1000 / 200 $\approx$ 5 éléments par liste en moyenne. En négligeant le temps de calcul du code et le temps d'accès par la table, on obtient **t_C** $\approx$ 5 k / 2 $\approx$ **2.5 k** .

Enfin, la structure **D** correspond à un accès dichotomique, le temps d'accès moyen à une feuille sera proportionnel à ($\log_2 1000$) soit **t_D** $\approx$ 10 k / 2 $\approx$**5k.**
Le classement demandé est donc : **C D B A .**

6.4. D'après la question précédente, on constate que c'est C qui présente le meilleur temps d'accès ; mais on a vu en **6.1.** qu'il ne permet pas d'imprimer l'index trié. La solution la plus adaptée au problème posé est donc la **solution D**, l'arbre binaire ordonné.



OUVRAGES GÉNÉRAUX

Ainsi naquit l'informatique ; R. Moreau
Principe des ordinateurs ; P. de Miribel
Emploi des ordinateurs ; J. C. Faure
Aide-mémoire d'informatique ; Ch. Berthet
Méthodes mathématiques de l'informatique; J. Vélu
L'informatique appliquée au calcul scientifique ; J.H. Saïac
D.P. words : dictionnaire d'informatique anglais-français, français-anglais ; G. Fehlmann
La sécurité informatique : approche méthodologique; J. M. Lamère
La sécurité des réseaux : méthodes et techniques ; J. M. Lamère et J. Tourly, Y. Leroux
Sécurité des petits et moyens systèmes ; J. M. Lamère, J. Tourly
Sûreté de fonctionnement des systèmes informatiques : matériels et logiciels ; J. C. Laprie, B. Courtois, M. C. Gaudel, D. Powell
Approche logique de l'intelligence artificielle, tomes 1 , 2 et 3 ; A. Thayse et co-auteurs.
Techniques de l'intelligence artificielle ; C. H. Dominé
Architecture et technologie des ordinateurs ; P. Zanella et Y. Ligier
Guide juridique de l'informatique ; B. Van Dorsselaere
L'anglais pour l'informatique ; L. Gallet et J. Brossard
Les systèmes informatiques : vision cohérente et utilisation ; C. Carrez

ALGORITHMIQUE, PROGRAMMATION

Initiation à l'analyse et à la programmation ; J. P. Laurent
Exercices commentés d'analyse et de programmation ; J. P. Laurent et J. Ayel
Les bases de la programmation ; J. Arsac
Raisonner pour programmer ; A. Gram
Proverbes de programmation ; H. F. Ledgard
Programmation, tomes 1 et 2 ; A. Ducrin
Théorie des programmes ; C. Livercy
Synchronisation de programmes parallèles ; F. André, D. Herman et J. P. Verjus
Algorithmique tomes 1 et 2 ; P. Berlioux et Ph. Bizard
Algorithmique du parallélisme ; M. Raynal
Initiation à l'algorithmique et aux structures de données, tomes 1, 2 et 3 ; J. Courtin et I. Kowarski
Construire les algorithmes ; C. Pair, R. Mohr et R. Schott
Parallélisme, génie logiciel temps réel ; M. Thorin
Programmation par syntaxe ; B. Groc et M. Bouhier

LANGAGES

La programmation en assembleur ; J. Rivière
Exercices d'assembleur et de macro-assembleur ; J. Rivière
Basic : programmation de micro-ordinateurs ; A. Checroun
Introduction à A.P.L. ; S. Pommier
Cobol : initiation et pratique ; M. Barès et H. Ducasse
Fortran IV ; M. Dreyfus
La pratique du Fortran ; M. Dreyfus et C. Gangloff
Fortran structuré et méthodes numériques ; S. Faroult et D. Simon
Le langage de programmation PL/1 ; Ch. Berthet
Prolog : fondements et applications ; M. Condillac
Pascal ISO/AFNOR : programmation déductive et description de la norme ; A. Tisserant
Turbo Pascal et son environnement ; J. Rivière
Le langage C ; D. Galland
Langages de quatrième génération ; Groupe LBD4G
Le langage ADA : manuel d'évaluation ; D. Le Verrand
ADA, un apprentissage ; M. Gauthier

BASES DE DONNÉES

Conception de bases de données ; Galacsi
Structures des bases de données ; M. Léonard
Structuration des données informatiques : initiation et applications ; B. Ibrahim et C. Pellegrini
Bases de données : méthodes pratiques sur maxi et mini-ordinateurs ; D. Martin
Techniques avancées pour bases de données ; D. Martin
Bases de données et systèmes relationnels ; C. Delobel et M. Adiba
Bases d'informations généralisées ; C. Chrisment, J.B. Crampes et G. Zurfluh
Les fichiers ; C. Jouffroy et C. Létang
Exploration informatique et statistique des données ; M. Jambu
Classification automatique des données ; G. Celeux, E. Diday et al.
Des structures aux bases de données ; C. Carrez

APPLICATIONS DE L'INFORMATIQUE

Les systèmes d'information : analyse et conception ; Galacsi
Comprendre les systèmes d'information ; Galacsi
Conception des systèmes d'information ; B. Herz, C. Moine et R. Sabatier
Systèmes informatiques répartis ; Cornafion
Systèmes d'exploitation des ordinateurs ; Crocus
Principes des systèmes d'exploitation des ordinateurs ; S. Krakowiak
Téléinformatique ; C. Macchi et J.F. Guilbert
Les réseaux de communication ; Neuvièmes Journées Francophones sur l'Informatique
Synthèse d'image : algorithmes élémentaires ; G. Hégron
Graphes, algorithmes et logiciels ; M. Minoux et G. Bartnik
De l'image à la décision : analyse des images numériques et théorie de la décision ; J. G. Postaire
Images de synthèse ; M. Bret
Logiciels interactifs et ergonomie ; M.F. Barthet
Le RNIS ; E. Iris
Télématique : techniques, normes, services ; B. Marti et al.
Méthodologies pour les systèmes d'information : guide de référence et d'évaluation ; T.W. Olle et al.

PROCESSEURS

Introduction aux microprocesseurs et aux micro-ordinateurs ; C. Pariot
Microprocesseurs : du 6800 au 6809, modes d'interfaçage ; G. Révellin
Interfaçage des microprocesseurs ; M. Robin et T. Maurin
Les architectures RISC ; J. C. Heudin et C. Panetto

INFORMATIQUE DE GESTION

Informatique de gestion : théorie, techniques, Cobol, pratique professionnelle ; Ch. Berthet
Merise 1 : méthode de conception ; A. Collongues, J. Hugues et B. Laroche
Merise 2 : études et exercices ; A. Collongues
Analyse organique, tomes 1 et 2 ; C. Cochet et A. Galliot
Merise : vers la conduite de projet ; B. Hugues, M. Grimal, B. Leblanc

Imprimerie GAUTHIER-VILLARS, Paris
Dépôt légal, Imprimeur, n° 3398

Dépôt légal : août 1990

Imprimé en France

INITIATION A L'ALGORITHMIQUE ET AUX STRUCTURES DE DONNÉES

3 - Problèmes, études de cas

Ce troisième tome a pour objectif de mettre en œuvre les principes et les méthodes algorithmiques développés dans les deux premiers : *1 - Programmation structurée et structures de données, et 2 - Récursivité et structures de données avancées.* Comme eux, il s'adresse aux étudiants de première année (DEUG, DEUST, DUT, BTS, MIAGE, MST, Licences, Ecoles d'ingénieurs...) et plus généralement à tous les lecteurs désireux de s'initier à la construction d'algorithmes corrects, qui sont la base de toute bonne programmation.

Les algorithmes sont exprimés dans un langage proche de Pascal. Très souvent, il sera demandé au lecteur de transformer ou d'adapter à des variables structurées, un algorithme connu utilisant des variables simples. Enfin, les exercices proposés, testés en travaux dirigés, sont de niveaux très divers afin que l'ensemble soit utile au plus grand nombre.



ISBN 2-04-019702-8

Didier Thimonier